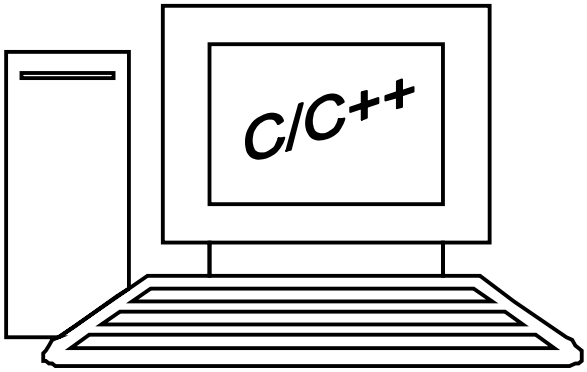
В.В. Войтенко, А.В. Морозов

C/C++

теорія та практика



МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
Житомирський державний технологічний університет

В. В. Войтенко
А. В. Морозов

C/C++ : Теорія та практика

Навчально-методичний посібник

Видання 2 – виправлене
(електронний варіант)

Житомир 2004

Навчально-методичний посібник для учнів та студентів різних форм навчання за професійним спрямуванням „Комп'ютерні науки".

Навч.-методичний посібник / В.В.Войтенко, А.В.Морозов. – Житомир: ЖДТУ, 2004. – 324 стор.

У першій частині навчального посібника викладено основи алгоритмічної мови Сі. На простих прикладах показано засоби її застосування до розв'язання практичних задач з програмування. Весь теоретичний виклад супроводжується прикладами. До розділу включено опис побудови алгоритмів у вигляді блок-схем, що може бути особливо корисне початківцям.

У другій частині посібника викладено основні положення об'єктно-орієнтованого підходу та способи його застосування до розв'язування задач з програмування за допомогою мови програмування Сі++.

Третя частина містить завдання для лабораторних та практичних робіт у різних варіантах. У додатках наведено прототипи найбільш широковживаних функцій мови Сі та Сі++.

Україна, Житомирський державний технологічний університет, 2004

Укладачі:

Войтенко Володимир Володимирович, кандидат технічних наук, доцент кафедри програмного забезпечення обчислювальної техніки ЖДТУ;

Морозов Андрій Васильович, студент ЖДТУ

Рецензент:

Панішев Анатолій Васильович, доктор технічних наук, професор, завідувач кафедри інформатики та математичного моделювання.

Затверджено на засіданні
вченої ради ЖІТІ,
протокол № 5 від 2 грудня 2002 р.

ПЕРЕДМОВА

Видання даного посібника було зумовлене зміною навчальних планів для підготовки студентів початкових курсів, що навчаються за спеціальностями 7.080403 "Програмне забезпечення автоматизованих систем" та 7.091401 "Системи управління і автоматики" у Житомирському інженерно-технологічному інституті. Починаючи з часів заснування факультету, в перші дисципліни загальної спеціалізації з програмування у вузівську програму незмінно включалася алгоритмічна мова Паскаль. Вона дійсно, у порівнянні з іншими мовами програмування високого рівня, найкраще підходила та й нині підходить для початкового ознайомлення студентів молодших курсів з основами алгоритмізації та програмування. Проте час багато що змінює: мова Паскаль стала обов'язковою частиною вивчення предмету „Інформатика" у старших класах середньої школи. Таким чином, переважна більшість вчорашніх школярів, приходячи до вузу на перший курс, вже має не лише початкові навички у програмуванні, а й відповідний чималий досвід програмування на Паскалі.

Виходячи з вищевказаних об'єктивних причин, замість вивчення основ програмування протягом першого семестру на прикладі мови Паскаль, було віддано перевагу мові Сі, яка раніше розглядалася починаючи з другого семестру. Перша частина даного посібника містить стислий, та водночас досить повний виклад мови Сі у відповідності до її стандарту ISO/IEC 14882. На простих прикладах показано засоби застосування мови для розв'язання практичних задач. Усі теоретичні відомості супроводжуються простими та зрозумілими прикладами. Викладення матеріалу за зростанням від простого до більш ускладненого допоможе краще зорієнтуватися тим студентам, хто знайомий з програмуванням на іншій мові. Крім того, до розділу включено опис усіх блочних мовних конструкцій побудови алгоритмів, що може бути особливо корисним початківцям.

Друга частина посібника (II семестр) присвячується мові Сі++, яка раніше розглядалася на старших курсах. На сьогодні об'єктно-орієнтоване програмування вже не є новою парадигмою, яка останнім часом зазнає суттєвих змін та модифікацій. На численних прикладах, наведених у другій частині посібника студенти ознайомляться з положеннями об'єктно-орієнтованої парадигми - абстрагуванням, інкапсуляцією, успадкуванням та засобами їх реалізації мовою Сі++.

У третьому розділі містяться завдання для лабораторних та практичних робіт згруповані за темами у варіантах. Сподіваємося, що студенти позитивно оцінять приблизно однакову їх складність, що виключить випадкову упередженість при виборі варіанту. У розділі

задач на складання ефективних алгоритмів згруповані завдання, які у попередні роки пропонувалися на шкільних та вузівських олімпіадах з програмування.

У додатках наведено прототипи найбільш широковживаних функцій мови Сі та Сі++, згруповані за належністю до стандартних бібліотек. При безпосередньому написанні програм цей розділ допоможе уникнути труднощів, пов'язаних з використанням довідників та вбудованого HELPа мови, особливо для тих, хто не достатньо володіє англійською мовою. Програмістам-практикам запропоновано велику кількість прикладів, що найкраще пояснюють ту чи іншу тему. Усі програмні фрагменти у посібнику уважно перевірені та відлагоджені, представляють собою так звані "консольні додатки", без прив'язки до конкретного операційного середовища. У даному посібнику не розглядається програмування під Windows та інші специфіковані середовища. За бажанням Ви можете отримати дискету, що містить усі програмні додатки, розміщені у посібнику.

У даному посібнику було виправлено вади та помилки попередніх видань, враховано побажання викладачів та студентів. Автори сподіваються на Ваші зауваження та побажання, які слід направляти за електронною адресою mav@zt.ukrtel.net. Вони будуть обов'язково враховані у подальшому.

Електронні версії посібників та інформацію про подальші видання кафедри програмного забезпечення обчислювальної техніки ЖІТІ можна знайти за адресою в Інтернеті: http://www.ziet.zhitomir.ua:8890/

ПРО АВТОРІВ

Войтенко Володимир Володимирович, кандидат технічних наук, доцент кафедри програмного забезпечення обчислювальної техніки ЖІТІ. У 1992 році закінчив Київський Національний університет ім. Тараса Шевченка. На кафедрі ПЗОТ працює з 1994 року. Викладає предмети "Основи програмування та алгоритмічні мови", "Сучасні технології програмування", "Об'єктно-орієнтоване проектування складних систем". E-mail : voytenko@ziet.zhitomir.ua

Морозов Андрій Васильович, у 2002 році закінчив міський ліцей при ЖІТІ, призер фінальних етапів *Всеукраїнських олімпіад і конкурсів у 2002 році*: WEB - олімпіади, учнівської олімпіади з інформатики, конкурсу науково-дослідницьких робіт Малої Академії Наук (відділення обчислювальної техніки та програмування), в даний час є студентом факультету інформаційно-комп'ютерних технологій ЖДТУ та тренером-викладачем Житомирського центру ІАТР. E-mail: morozov@iatp.org.ua, mav@zt.ukrtel.net.

ЧАСТИНА 1. МОВА ПРОГРАМУВАННЯ СІ

1.1 Історія виникнення

Трохи про історію виникнення мов програмування, та мови Сі зокрема. У 1949 році у Філадельфії (США) під керівництвом Джона Мочлі був створений "Стислий код" – перший примітивний інтерпретатор мови програмування. У 1951 році у фірмі Remington Rand американська програмістка Грейс Хоппер розробила першу транслюючи програму, що називалася компілятором (compiler – компоновщик). У 1957 році у штаб-квартирі фірми IBM на Медісон-авеню у Нью-Йорку з'явилася перша повна мова Фортран (FORmula TRANslation – трансляція формул). Групою розробників керував тоді відомий 30-річний математик Джон Бекус. Фортран – це перша із "дійсних" мов високого рівня.

Далі, у 1972 році 31-літній фахівець із системного програмування фірми Bell Labs Денніс Рітчі розробив мову програмування Сі. У 1984 році французький математик та саксофоніст Филип Кан засновує фірму Borland International. Далі з'явився діалект мови Сі фірми Borland.

На початку Сі була розроблена як мова для програмування в операційній системі Unix. Незабаром він став поширюватися для програмістів-практиків. Наприкінці 70-х були розроблені транслятори Сі для мікроЕОМ операційної системи CP/M. Після появи IBM PC стали з'являтися і компілятори мови Сі (для таких комп'ютерів їх зараз декілька десятків). У 1983 р. американський Інститут Стандартів (ANSI) сформував Технічний Комітет X3J11 для створення стандарту мови Сі. На сьогодні мова Сі++, що з'явилася як послідовник Сі, підпорядковується більшості вимог стандарту.

За своїм змістом Сі, перш за все, є мовою функцій. Програмування на Сі здійснюється шляхом опису функцій і звертання до бібліотек (бібліотечних функцій). Більшість функцій повертають деякі значення, що можуть використовуватися в інших операторах.

Серед переваг мови Сі потрібно відзначити основні:

- універсальність (використовується майже на всіх існуючих ЕОМ);
- компактність та універсальність коду;
- швидкість виконання програм;
- гнучкість мови;
- висока структурованість.

1.2 Елементи мови Сі

Будь-яка мова (українська, російська, англійська, французька та інші) складається з декількох основних елементів – символів, слів, словосполучень і речень. В алгоритмічних мовах програмування існують аналогічні структурні елементи, тільки слова називають лексемами, словосполучення – виразами, а речення – операторами.

Лексеми в свою чергу утворюються із символів, вирази – із лексем і символів, оператори – із символів, лексем і виразів.

- *Алфавіт мови*, або її символи – це основні неподільні знаки, за допомогою яких пишуться всі тексти на мові програмування.
- *Лексема*, або елементарна конструкція – мінімальна одиниця мови, яка має самостійний зміст.
- *Вираз* задає правило обчислення деякого значення.
- *Оператор* задає кінцевий опис деякої дії.

1.2.1 Алфавіт

Алфавіт мови Сі включає :

- великі та малі літери латинської абетки;
- арабські цифри;
- пробільні символи : пробіл, символи табуляції, символ переходу на наступний рядок тощо;
- символи , . ; : ? ‘ ! | / \ ~ ( ) [ ] { } < > # % ^ & – + * =

1.2.2 Ідентифікатори

Ідентифікатори використовуються для іменування різних об’єктів : змінних, констант, міток, функцій тощо. При записі *ідентифікаторів* можуть використовуватися великі та малі літери латинської абетки, арабські цифри та символ підкреслення. Ідентифікатор не може починатися з цифри і не може містити пробілів.

Компілятор мови Сі розглядає літери верхнього та нижнього регістрів як різні символи. Тому можна створювати ідентифікатори, які співпадають орфографічно, але відрізняються регістром літер. Наприклад, кожний з наступних ідентифікаторів унікальний :

```
Sum   sum   sUm   SUM   sUM
```

Слід також пам’ятати, що ідентифікатори не повинні співпадати з ключовими словами.

1.2.3 Константи

Константами називають сталі величини, тобто такі, які в процесі виконання програми не змінюються. В мові Сі існує чотири типи констант : цілі, дійсні, рядкові та символьні.

1. Цілі константи можуть бути десятковими, вісімковими або шістнадцятковими.

Десяткова константа – послідовність десяткових цифр (від 0 до 9), яка починається не з нуля якщо це число не нуль. Приклади десяткових констант : 10, 132, 1024.

Вісімкові константи починаються з символу 0, після якого розміщуються вісімкові цифри (від 0 до 7). Наприклад : 023. Запис константи вигляду 08 буде сприйматися компілятором як помилка, так як 8 не є вісімковою цифрою.

Шістнадцяткові константи починаються з символів 0x або 0X, після яких розміщуються шістнадцяткові цифри (від 0 до F, можна записувати їх у верхньому чи нижньому регістрах). Наприклад : 0XF123.

2. Дійсні константи складаються з цілої частини, десяткової крапки, дробової частини, символу експоненти (e чи E) та показника степеня. Дійсні константи мають наступний формат представлення :

[ціла_частина][. дробова_частина][E [–] степінь]

У записі константи можуть бути опущені ціла чи дробова частини (але не обидві разом), десяткова крапка з дробовою частиною чи символ E (e) з показником степеня (але не разом). Приклади дійсних констант : 2.2 , 220e–2, 22.E–1, .22E1.

Якщо потрібно сформувати від'ємну цілу або дійсну константу, то перед константою необхідно поставити знак унарного мінуса.

3. Символьні константи. Символьна константа – це один або декілька символів, які заключені в апострофи. Якщо константа складається з одного символу, вона займає в пам'яті 1 байт (тип *char*). Двосимвольні константи займають в пам'яті відповідно 2 байти (тип *int*).

Послідовності символів, які починаються з символу \ (зворотний слеш) називаються керуючими або escape-послідовностями (таблиця 1.1).

Таблиця 1.1. Escape-послідовності

Спеціальний символ	Шістнадцят-ковий код	Значення
\a	07	звуковий сигнал
\b	08	повернення на 1 символ
\f	0C	переведення сторінки
\n	0A	перехід на наступний рядок
\r	0D	повернення каретки
\t	09	горизонтальна табуляція
\v	0B	вертикальна табуляція
\\	5C	символ \
\’	27	символ ‘
\”	22	символ “
\?	3F	символ ?
\0	00	нульовий символ
\0ddd	–	вісімковий код символу
\0xddd	ddd	десятковий код символу

4. Рядкові константи записуються як послідовності символів, заключених в подвійні лапки.

```
"Це рядковий літерал!\n"
```

Для формування рядкових констант, які займають декілька рядків тексту програми використовується символ \ (зворотний слеш):

```
"Довгі рядки можна розбивати на \
 частини"
```

Загальна форма визначення іменованої константи має вигляд :

const тип ім’я = значення ;

Модифікатор *const* попереджує будь-які присвоювання даному об’єкту, а також інші дії, що можуть вплинути на зміну значення. Наприклад:

```
const float  pi = 3.1415926;
const maxint = 32767;
char *const str="Hello,P...!"; /* покажчик-константа */
char const *str2= "Hello!"; /* покажчик на константу */
```

Використання одного лише модифікатору *const* еквівалентно *const int*.

1.2.4 Коментарі

Текст на Сі, що міститься у дужках /* та */ ігноруватиметься компілятором, тобто вважатиметься коментарем до програми. Такі коментарі можуть розміщуватися в будь-якому місці програми.

Коментарі здебільшого використовуються для „документування програм" та під час їх відлагодження. В програму бажано вміщувати текст, що хоч якось пояснює її роботу та призначення. Проте не слід надто зловживати коментарями, а використовувати більш розумні форми найменування змінних, констант, функцій тощо. Якщо, наприклад, функція матиме назву *add_matrix*, очевидно не зовсім раціональним буде включення у програму після її заголовної частини коментар про те, що:

```
/*функція обчислює суму матриць */
```

У цьому випадку ім'я функції пояснює її призначення. У більш сучасних версіях Сі широко застосовується так званий угорський запис імен, коли ім'я змінної містить в собі інформацію про її призначення і тип.

1.2.5 Ключові слова

Ключові слова – це зарезервовані ідентифікатори, які мають спеціальне значення для компілятора. Їх використання суворо регламентоване. Імена змінних, констант, міток, типів тощо не можуть співпадати з ключовими словами.

Наводимо перелік ключових слів мови Сі :

auto	***continue***	***float***	***interrupt***	***short***	***unsigned***
asm	***default***	***for***	***long***	***signed***	***void***
break	***do***	***far***	***near***	***sizeof***	***volatile***
case	***double***	***goto***	***pascal***	***static***	***while***
cdecl	***else***	***huge***	***switch***	***struct***	
char	***enum***	***if***	***register***	***typedef***	
const	***extern***	***int***	***return***	***union***	

1.3 Структура програми. Базові типи даних.

1.3.1 Функція main() : з цього все починається

Усі програми, написані на мові *Ci*, повинні містити в собі хоча б одну функцію. Функція *main()* – вхідна точка будь-якої програмної системи, причому немає різниці, де її розміщувати. Але потрібно

пам'ятати наступне: якщо вона буде відсутня, завантажувач не зможе зібрати програму, про що буде виведене відповідне попередження. Перший оператор програми повинен розміщуватися саме в цій функції.

Мінімальна програма на мові Сі має вигляд:

```
main()
{
  return 0;
}
```

Функція починається з імені. В даному прикладі вона не має параметрів, тому за її ім'ям розташовуються порожні круглі дужки *()*. Далі обидві фігурні дужки {...} позначають блок або складений оператор, з яким ми працюватимемо, як з єдиним цілим. У Паскалі аналогічний зміст мають операторні дужки *begin ... end*.

Мінімальна програма має лише один оператор - оператор повернення значення *return*. Він завершує виконання програми та повертає в нашому випадку деяке ціле значення (ненульове значення свідчить про помилки в програмі, нульове про успішне її завершення). Виконання навіть цієї найпростішої програми, як і решти багатьох, проходить у декілька етапів (рис 1.1.) :

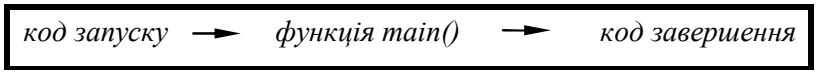


Рис. 1.1. Етапи виконання програми на мові Сі

1.3.2 Базові типи даних

Будь-яка програма передбачає виконання певних операцій з даними. Від їх типу залежить, яким чином будуть проводитися ці операції, зрештою, буде визначено, як реалізовуватиметься алгоритм.

Що таке тип даних? Сформулювати це поняття можна так : множина значень плюс перелік дій або операцій, які можна виконати над кожною змінною даного типу. Вважається, що змінна або вираз належить до конкретного типу, якщо його значення лежить в області допустимих значень цього типу.

Арифметичні типи даних об'єднують цілі та дійсні, цілі у свою чергу - декілька різновидів цілих та символьних типів даних. Скалярні типи включають в себе арифметичні типи, покажчики та перелічувані типи. Агрегатні або структуровані типи містять в собі масиви, структури та файли. Нарешті функції представляють дещо особливий клас, який слід розглядати окремо.

Базові типи даних *Ci* можна перерахувати у наступній послідовності:

1. char – символ

Тип може використовуватися для зберігання літери, цифри або іншого символу з множини символів ASCII. Значенням об'єкта типу *char* є код символу. Тип *char* інтерпретується як однобайтове ціле з областю значень від –128 до 127.

2. int – ціле

Цілі числа у діапазоні від –32768 до 32767. В операційних середовищах Windows та Windows NT використовуються 32-розрядні цілі, що дозволяє розширити діапазон їх значень від –2147483648 до 2147483647. Як різновиди цілих чисел, у деяких версіях компіляторів існують *short* - коротке ціле (слово) та *long* (4 байти) - довге ціле. Хоча синтаксис мови не залежить від ОС, розмірність цих типів може коливатися від конкретної реалізації. Гарантовано лише, що співвідношення розмірності є наступним: $short \le int \le long$.

3. float – число з плаваючою комою одинарної точності

Тип призначений для зберігання дійсних чисел. Може представляти числа як у фіксованому форматі (наприклад число пі - 3.14159), так і в експоненціальній формі – 3.4E+8.

4. double - число з плаваючою комою подвійної точності

Має значно більший діапазон значень, порівняно з типом *float*: $\pm(1.7\cdot10^{-308} \ldots 1.7\cdot10^{308})$.

У мові Сі, на відміну від Паскаля, використовується префіксний запис оголошення. При цьому на початку вказується тип змінної, а потім її ім'я. Змінні повинні бути описаними до того моменту, як вони будуть використовуватися у програмі. Ніяких додаткових ключових слів при цьому не пишуть. Наприклад:

```
int name;
float var, var1;
double temp;
char ch;
long height;
```

Змінні можна ініціалізувати (присвоювати їм початкові значення) безпосередньо у місці їх опису:

```
int height = 33 ;
float income = 2834.12 ;
char val = 12 ;
```

Для виведення інформації на екран використаємо функцію *printf()* (детально про операції введення-виведення значень змінних йтиметься у розділі 1.3.4. "Функції введення та виведення"):

```
printf("Вік Олега-%d.Його прибуток %.2f",age,income);
```

Крім того, цілі типи *char*, *short*, *int*, *long* можуть використовуватися з модифікаторами *signed* (із знаком) та *unsigned* (без знаку). Цілі без знаку (*unsigned*) не можуть набувати від'ємних значень, на відміну від знакових цілих (*signed*). За рахунок цього дещо розширюється діапазон можливих додатних значень типу (таблиця 1.2.).

Таблиця 1.2. Діапазони значень простих типів даних

Тип	Діапазон значень	Розмір (байт)
char	-128 … 127	1
short	-32768 ... 32767	2
int		2 або 4
long	-2,147,483,648 ... 2,147,483,647	4
unsigned char	0 ... 255	1
unsigned short	0 … 65535	2
unsigned		2 або 4
unsigned long	0 ... 4,294,967,295	4
float	$\pm(3.4\cdot10^{-38}$ ... $3.4\cdot10^{38})$	4
double	$\pm(1.7\cdot10^{-308}$ ... $1.7\cdot10^{308})$	8
long double	$\pm(3.4\cdot10^{-4932}$ ... $3.4\cdot10^{4932})$	10

1.3.3 Перетворення типу

Згадаємо, що компілятор Паскаля виконує автоматичне перетворення типів даних, особливо в математичних виразах, коли найчастіше цілочисельний тип перетворюється у тип з плаваючою комою. Цей стиль підтримує і Сі, причому значення типу *char* та *int* в арифметичних виразах змішуються: кожний з таких символів автоматично перетворюється в ціле. Взагалі, якщо операнди мають

різні типи, перед тим, як виконати операцію, молодший тип "підтягується" до старшого. Результат - старшого типу. Отже,

- *char* та *short* перетворюються в *int*;
- *float* перетворюється в *double*;
- якщо один з операндів *long double*, то і другий перетворюється в *long double*;
- якщо один з операндів *long*, тоді другий перетворюється відповідно до того ж типу, і результат буде *long*;
- якщо один з операндів *unsigned*, тоді другий перетворюється відповідно до того ж типу, і результат буде *unsigned*.

Приклад:

```
double ft, sd;
unsigned char ch;
unsigned long in;
int i;
/* ... */
sd = ft*(i+ch/in);
```

При виконанні оператора присвоювання в даному прикладі правила перетворення типів будуть використані наступним чином. Операнд *ch* перетворюється до *unsigned int*. Після цього він перетворюється до типу *unsigned long*. За цим же принципом *i* перетворюється до *unsigned long* і результат операції, що розміщена в круглих дужках буде мати тип *unsigned long*. Потім він перетворюється до типу *double* і результат всього виразу буде мати тип *double*.

Взагалі, тип результату кожної арифметичної операції виразу є тип того операнду, який має у відповідності більш високий тип приведення.

Але, окрім цього в Сі, з'являється можливість і примусового перетворення типу, щоб дозволити явно конвертувати (перетворювати) значення одного типу даних в інший. Загальний синтаксис перетворення типу має два варіанти :

1). (новий_тип) вираз ;

2). новий_тип (вираз) ;

Обидва варіанти перетворення виглядають так:

```
char letter = 'a';
int nasc = int (letter);
long iasc = (long) letter;
```

1.3.4 Функції введення та виведення

Що б там не було, але реальні програми важко уявити без використання операцій введення та виведення.

В мові Сі на стандартні потоки введення-виведення (в більшості випадків – клавіатура та монітор) завжди вказують імена *stdin* та *stdout*. Обробку цих потоків здійснюють функції, визначені в заголовочному файлі *stdio.h*.

Розглянемо основні функції введення-виведення.

Функція *getchar()* зчитує і повертає черговий символ з послідовності символів вхідного потоку. Якщо цю послідовність вичерпано, то функція *getchar()* повертає значення –1 (цьому значенню відповідає константа *EOF*).

Функція *putchar(аргумент)*, де аргументом є вираз цілого типу, виводить у стандартний вихідний потік значення аргументу, перетворене до типу *char*.

Приклад :

```
#include<stdio.h>
void main()
{
  char ch;
  ch=getchar();
  putchar(ch);
}
```

Для введення та виведення більш складної інформації використовуються функції *scanf()* та *printf()*.

Функція *printf()* призначена для виведення інформації за заданим форматом. Синтаксис функції *printf()*:

printf(“Рядок формату”[, аргумент1[, аргумент2, [...]]]);

Першим параметром даної функції є „рядок формату”, який задає форму виведення інформації. Далі можуть розташовуватися вирази арифметичних типів або рядки (в списку аргументів вони відокремлюються комами). Функція *printf()* перетворює значення аргументів до вигляду, поданого у рядку формату, „збирає” перетворені значення в цей рядок і виводить одержану послідовність символів у стандартний потік виведення.

Рядок формату складається з об’єктів двох типів : звичайних символів, які з рядка копіюються в потік виведення, та специфікацій перетворення. Кількість специфікацій у рядку формату повинна дорівнювати кількості аргументів.

Таблиця 1.3. Значення основних модифікаторів рядка формату

Модифікатор	Значення
–	Аргумент буде друкуватися починаючи з лівої позиції поля заданої ширини. Звичайно друк аргументу закінчується в самій правій позиції поля. Приклад : %-10d
Рядок цифр	Задає мінімальну ширину поля. Поле буде автоматично збільшуватися, якщо число або рядок не буде вміщуватися у полі. Приклад : %4d
Цифри.цифри	Визначає точність : для типів даних з плаваючою комою - число символів, що друкуються зліва від десяткової коми; для символьних рядків – максимальну кількість символів, що можуть бути надруковані. Приклад : %4.2f

Приклад :

```
#include<stdio.h>
void main()
{
   int a=10,b=20,c=30;
   printf(" a==%d \n b==%d \n c==%d \n",a,b,c);
}
```

Специфікації перетворення для функції *printf()*:

%d – десяткове ціле;
%i – десяткове ціле;
%o – вісімкове ціле без знаку;
%u – десяткове ціле без знаку (unsigned)
%x – шістнадцяткове ціле без знаку;
%f – представлення величин float та double з фіксованою точкою;
%e або %E – експоненціальний формат представлення дійсних величин;
%g – представлення дійсних величин як f або E в залежності від значень;
%c – один символ (char);
%s – рядок символів;
%p – покажчик
%n – покажчик
%ld – long (в десятковому вигляді);
%lo – long (у вісімковому вигляді);
%p – виведення покажчика в шістнадцятковій формі;
%lu – unsigned long.

Можна дещо розширити основне визначення специфікації перетворення, помістивши модифікатори між знаком % і символами, які визначають тип перетворення (таблиця 1.3.).

Розглянемо декілька прикладів:

Приклад 1 :

```
#include <stdio.h>
main()
{
  printf("/%d/\n",336);
  printf("/%2d/\n",336);
  printf("/%10d/\n",336);
  printf("/%-10d/\n",336);
};
```

Результат виконання програми буде виглядати так :

```
/336/
/336/
/       336/
/336         /
```

Приклад 2 :

```
#include <stdio.h>
main()
{
  printf("/%f/\n",1234.56);
  printf("/%e/\n",1234.56);
  printf("/%4.2f/\n",1234.56);
  printf("/%3.1f/\n",1234.56);
  printf("/%10.3f/\n",1234.56);
  printf("/%10.3e/\n",1234.56);
}
```

На цей раз результат виконання програми буде виглядати так :

```
/1234.560000/
/1.234560e+03/
/1234.56/
/1234.6/
/  1234.560/
/ 1.235e+03/
```

Для введення інформації зі стандартного потоку введення використовується функція *scanf().*

Синтаксис :

scanf("Рядок формату",&аргумент1[,&аргрумент2[, ...]]);

Так, як і для функції *printf(),* для функції *scanf()* вказується рядок формату і список аргументів. Суттєва відмінність у синтаксисі цих двох функцій полягає в особливостях даного списку аргументів.

Функція *printf()* використовує імена змінних, констант та вирази, в той час, як для функції *scanf* () вказується тільки покажчики на змінні.

Поширеною помилкою використання *scanf()* у початківців є звертання: *scanf("%d",n)* замість *scanf("%d",&n).* Параметри цієї функції обов'язково повинні бути покажчиками!

Функція *scanf()* використовує практично той же набір символів специфікації, що і функція *printf()*.

```
#include <stdio.h>
main()
{
  int a,b,c;
  printf("A=");
  scanf("%d",&a);
  printf("B=");
  scanf("%d",&b);
  c=a+b;
  printf("A+B=%d",c);
}
```

Більшість реалізацій мови Сі дозволяють пов'язувати імена *stdin* та *stdout* не тільки з клавіатурою та екраном, а й із зовнішніми файлами. Для цього в рядку виклику Сі програми необхідно вказати імена цих файлів. Якщо перед ім'ям файлу введення поставити знак <, то даний файл буде пов'язаний з потоком введення.

```
prog < file.in
```

В даному прикладі інформація читається з файлу *file.in* поточного каталогу, а не з клавіатури, тобто цей файл стає стандартним файлом введення, на який вказує *stdin*.

```
prog > file.out
```

А при такому виклику програми інформація виводиться не на екран, а у файл *file.out*.

Якщо необхідно читати інформацію з одного файлу, а результати записувати у інший одразу, виклик програми буде мати вигляд :

```
prog < file.in > file.out
```

1.3.5 Директиви включення

У багатьох програмах ми зустрічаємо використання так званих директив включення файлів. Синтаксис використання їх у програмі наступний :

```
# include <file_1>
# include <file_2>
...
# include <file_n>
```

По-перше, слід звернути увагу на те, що на відміну від більшості операторів, ця директива не завершується крапкою з комою. Використання таких директив призводить до того, що препроцесор підставляє на місце цих директив тексти файлів у відповідності з тими, що перелічені у дужках < ... > . Якщо ім'я файла міститься у таких дужках, то пошук файлу буде проводитися у спеціальному каталозі файлів для включення (як, правило, каталог *INCLUDE*, усі файли з розширенням **.h - header-файли*). Якщо даний файл у цьому каталозі буде відсутнім, то препроцесор видасть відповідне повідомлення про помилку, яка є досить типовою для початківців при роботі в інтегрованому середовищі:

< Unable to open include file 'file.h'. >

<Неможливо відкрити файл включення ' file.h'>

У цьому випадку достатньо перевірити не тільки наявність *header-файлу* у відповідній директорії, але й впевнитися у тому, що опція *Options\Directories* дійсно відповідає правильному диску та спеціальному каталогу, де розташовані файли включення.

Існує і другий спосіб - вказівка імені файлу у подвійних лапках - *"file_n.txt* ", так найчастіше підключають програмісти власноруч створені файли включення. Тоді пошук файлу ведеться у поточній директорії активного диску, якщо ж пошук буде невдалим, система закінчує його у спеціальному каталозі для *header*-файлів, як і у загальному випадку. Найбільш частим у початківців є включення файлу *"stdio.h"*:

```
#include <stdio.h>
main()
{
  printf("Hello ! ...\n");
  return 0;
}
```

Слід зауважити, що файли включення іноді можуть вміщувати в собі командні рядки включення інших файлів, причому без ніяких обмежень у глибину вкладеності. Цей прийом широко застосовується при розробці великих програмних проектів.

1.4 Основні операції

Операції подібні вбудованим функціям мови програмування. Вони застосовуються до виразів (операндів). Більшість операцій мають два операнди, один з яких розташовується перед знаком операції, а інший – після. Наприклад, два операнди має операція додавання A+B. Операції, які мають два операнди називаються *бінарними*. Існують і *унарні* операції, тобто такі, які мають лише один операнд. Наприклад, запис –A означає застосування до операнду A операції унарного мінуса. А три операнди має лише одна операція – ?:. Це єдина *тернарна* операція мови Сі.

У складних виразах послідовність виконання операцій визначається дужками, старшинством операцій, а при однаковому старшинстві – асоціативністю.

За призначенням операції можна поділити на :

- арифметичні операції;
- операції присвоювання;
- операції відношення;
- логічні операції;
- порозрядні операції;
- операція обчислення розміру *sizeof()*;
- умовна операція ?;
- операція слідування (кома).

1.4.1 Арифметичні операції

До арифметичних операцій належать відомі всім бінарні операції додавання, віднімання, множення, ділення та знаходження залишку від ділення (таблиця 1.4.).

Таблиця 1.4. Бінарні арифметичні операції

Операція	Значення	Приклад
+	Додавання	a+b
–	Віднімання	a–b
*	Множення	a*b
/	Ділення	a/b
%	Залишок від ділення	a%6

Для наведених арифметичних операцій діють наступні правила :

- бінарні операції додавання (+) та віднімання (–) можуть застосовуватися до цілих та дійних чисел, а також до покажчиків;
- в операціях множення (*) та ділення (/) операнди можуть бути будь-яких арифметичних типів;
- операція „залишок від ділення” застосовується лише до цілих операндів.
- Операції виконуються зліва направо, тобто спочатку обчислюється вираз лівого операнда, потім вираз, що стоїть справа від знака операції. Якщо операнди мають однаковий тип, то результат арифметичної операції має той же тип. Тому, коли операції ділення / застосовується до цілих або символьних змінних, залишок відкидається. Так, вираз 11/3 буде рівний 3, а вираз 1/2 буде рівним нулю.

В мові Сі визначені також і унарні арифметичні операції (таблиця 1.5.).

Операція інкременту (++) збільшує операнд на одиницю, а операція декременту (--) відповідно зменшує операнд на одиницю. Ці операції виконуються швидше, ніж звичайні операції додавання одиниці (a=a+1;) чи віднімання одиниці (a=a–1;).

Таблиця 1.5. Унарні арифметичні операції

Операція	*Значення*	*Приклад*
+	Унарний плюс (підтвердження знака)	+5
–	Унарний мінус (зміна знака)	–x
++	Операція інкременту (збільшення на 1)	i++, ++i
—	Операція декременту (зменшення на 1)	j—, —j

Існує дві форми запису операцій інкременту та декременту : префіксна та постфіксна.

Якщо операція інкременту (декременту) розміщена перед змінною, то говорять про префіксну форму запису інкременту (декременту). Якщо операція інкременту (декременту) записана після змінної, то говорять про постфіксну форму запису. У префіксній формі змінна спочатку збільшується (зменшується) на одиницю, а потім її нове значення використовується у виразі. При постфіксній формі у виразі спочатку використовується поточне значення змінної, а потім відбувається збільшення (зменшення) цієї змінної на одиницю.

Приклад, який демонструє роботу операції інкременту:

```
#include<stdio.h>
void main()
{
  int x=3,y=3;
  printf("Значення префіксного виразу : %d\n ",++x);
  printf("Значення постфіксного виразу: %d\n ",y++);
  printf("Значення x після інкременту : %d\n ",x);
  printf("Значення y після інкременту : %d\n ",y);
}
```

1.4.2 Операції присвоювання

В мові Сі знак = не означає „дорівнює". Він означає операцію присвоювання деякого значення змінній. Тобто зміст рядка вигляду *„vr1=1024;"* не виражається словами *„vr1 дорівнює 1024"*. Замість цього потрібно казати так : *„присвоїти змінній vr1 значення 1024"*.

Перелік операцій присвоювання мови Сі ілюструє таблиця 1.6.

Операція присвоювання повертає як результат присвоєне значення. Завдяки цьому в мові Сі допускаються присвоювання виду :

```
a=(b=c=1)+1;
```

Розглянемо приклад, який демонструє використання таких присвоювань.

Таблиця 1.6. Операції присвоювання

Операція	*Значення*
a = b	присвоювання значення *b* змінній *a*
a += b	додавання з присвоюванням. Означає a = a + b
a –= b	віднімання з присвоюванням. Означає a = a – b
a *= b	множення з присвоюванням. Означає a = a * b
a /= b	ділення з присвоюванням. Означає a = a / b
a %= b	залишок від ділення з присвоюванням. Означає a = a % b
a <<= b	зсув вліво з присвоюванням. Означає a = a << b
a >>= b	зсув вправо з присвоюванням. Означає a = a >> b
a &= b	порозрядне І з присвоюванням. Означає a = a & b
a |= b	порозрядне АБО з присвоюванням. Означає a = a | b
a ^= b	побітове додавання за МОД2 з присвоюванням, означає a = a ^ b

```
#include<stdio.h>
void main()
{
  int data1, data2, data3;
  data1=data2=data3=68;
  printf("\ndata1==%d\ndata2==%d\ndata3==%d",
                                    data1,data2,data3);
}
```

Результат роботи програми виглядає так :

```
data1==68
data2==68
data3==68
```

data1=data2=data3=68;

Присвоювання відбувається справа наліво : спочатку змінна *data3* отримує значення 68, потім змінна *data2* і нарешті *data1*.

1.4.3 Операції порівняння

Операції порівняння здебільшого використовуються в умовних виразах. Приклади умовних виразів :

b<0, ‘b’==’B’,’f’!=’F’, 201>=205,

Кожна умова перевіряється : істинна вона чи хибна. Точніше слід сказати, що кожна умова приймає значення „істинно” (true) або “хибно” (flase). В мові Сі немає логічного (булевого) типу. Тому результатом умовного виразу є цілочисельне арифметичне значення. „Істинно” – це ненульова величина, а „хибно” – це нуль. В більшості випадків в якості ненульового значення „істинно” використовується одиниця.

Приклад :

```
#include<stdio.h>
main()
{
  int tr, fal;
  tr=(111<=115); /* вираз істинний */
  fal=(111>115); /* вираз хибний */
  printf("true – %d false – %d \n",tr,fal);
  return 0;
}
```

Таблиця 1.7. Операції порівняння

Операція	*Значення*
<	Менше
<=	менше або рівно
==	перевірка на рівність
>=	більше або рівно
>	Більше
!=	перевірка на нерівність

1.4.4 Логічні операції

Логічні операції &&, ||, ! використовуються здебільшого для „об'єднання" виразів порівняння у відповідності з правилами логічного І, логічного АБО та логічного заперечення (таблиця 1.8.).

Таблиця 1.8. Логічні операції

Операція	*Значення*
&&	логічне І (and)
||	логічне АБО (or)
!	логічне заперечення (not)

Складні логічні вирази обчислюються „раціональним способом". Наприклад, якщо у виразі

```
(A<=B)&&(B<=C)
```

виявилось, що А більше В, то всі вирази, як і його перша частина (A<=B), приймають значення „хибно", тому друга частина (B<=C) не обчислюється.

Результат логічної операції 1, якщо істина і 0 у протилежному випадку.

Таблиця 1.9. Таблиця істинності логічних операцій

E1	E2	E1&&E2	E1||E2	!E1
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

1.4.5 Порозрядні операції (побітові операції)

Порозрядні операції застосовуються тільки до цілочисельних операндів і „працюють" з їх двійковими представленнями. Ці операції неможливо використовувати із змінними типу *double*, *float*, *long double*.

Таблиця 1.10. Порозрядні операції

Операція	*Значення*
~	порозрядне заперечення
&	побітова кон'юнкція (побітове І)
|	побітова диз'юнкція (побітове АБО)
^	побітове додавання за МОД2
<<	зсув вліво
>>	зсув вправо

Таблиця 1.11. Таблиця істинності логічних порозрядних операцій

E1	E2	E1&E2	E1^E2	E1|E2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

- Порозрядне заперечення ! заміняє змінює кожну 1 на 0, а 0 на 1. Приклад : ~ (10011010) == (01100101).
- Порозрядна кон'юнкція & (порозрядне І) порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо тільки два відповідних розряди операндів рівні 1, в інших випадках результат 0.
- Приклад : (10010011) & (00111101) == (00010001).
- Порозрядна диз'юнкція | (порозрядне АБО) порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо хоча б один з відповідних розрядів рівний 1.
  Приклад : (10010011) | (00111101) == (10111111)

- Побітове додавання за МОД2 порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо один з двох (але не обидва) відповідних розряди рівні 1.
  Приклад : (10010011) ^ (00111101) == (10101110)
  На операції побітового додавання за МОД2 грунтується метод обміну значень двох цілочисельних змінних.

```
a^=b^=a^=b;
```

- Операція зсуву вліво (вправо) переміщує розряди першого операнду вліво (вправо) на число позицій, яке задане другим операндом. Позиції, що звільняються, заповнюються нулями, а розряди, що зсуваються за ліву (праву) границю, втрачаються.
  Приклади :
  (10001010) << 2 == (00101000)
  (10001010) >> 2 == (00100010)

1.4.6 Операція слідування (кома)

Операція „кома" (,) називається операцією слідування, яка „зв'язує" два довільних вирази. Список виразів, розділених між собою комами, обчислюються зліва направо. Наприклад, фрагмент тексту

```
a=4;
b=a+5;
```

можна записати так :

```
a=4, b=b+5;
```

Операція слідування використовується в основному в операторах циклу *for()* (про оператори циклів піде мова пізніше).

Для порівняння наводимо приклад з використанням операції слідування (приклад 2) та без неї (приклад 1):

Приклад 1.

```
int a[10],sum,i;
…
sum=a[0];
for (i=1;i<10;i++)
        sum+=a[i];
```

Приклад 2.

```
int a[10],sum,i;
/* … */
for (i=1,sum=a[0];i<10;sum+=a[i],i++) ;
```

1.4.7 Умовна операція ?:

Умовна операція ?: – єдина тернарна операція в мові Сі. Її синтаксис :

умова ? вираз_1 : вираз_2

Принцип її роботи такий. Спочатку обчислюється вираз умови. Якщо цей вираз має ненульове значення, то обчислюється *вираз_1*. Результатом операції ?: в даному випадку буде значення *виразу_1*. Якщо вираз умови рівний нулю, то обчислюється *вираз_2* і його значення буде результатом операції. В будь-якому випадку обчислюється тільки один із виразів (*вираз_1* або *вираз_2*).

Наприклад, дану операцію зручно використати для знаходження найбільшого з двох чисел *x* і *y*:

```
max=(x>y)?x:y;
```

Приклад 1 :

```
#include<stdio.h>
void main()
{
  int points;
  printf("Введіть оцінку [2..5]:");
  scanf("%d",&points);
  printf("%s",points>3?"Ви добре знаєте
матеріал!":"Погано...");
}
```

Приклад 2 :

```
j = (i<0) ? (-i) : (i); /* змінній j присвоюється
модуль i*/
```

1.4.8 Операція sizeof()

Дана операція обчислює розмір пам’яті, необхідний для розміщення в ній виразів або змінних вказаних типів.

Операція має дві форми :

1). ім’я_типу A;
    sizeof A;
2). sizeof (ім’я_типу);

Операцію *sizeof*() можна застосовувати до констант, типів або змінних, у результаті чого буде отримано число байт, що відводяться під операнд. Приміром, *sizeof*(*int*) поверне число байт для розміщення змінної типу *int*.

1.5 Основи алгоритмізації

1.5.1 Алгоритми та їх властивості

Алгоритм – це чітко визначена для конкретного виконавця послідовність дій, які спрямовані на досягнення поставленої мети або розв'язання задачі певного типу.

У 820 році нашої ери в Бухарі був написаний підручник „Аль-Джабр Ва-аль-Мукабала" („Наука виключення скорочення"), в якому були описані правила виконання чотирьох арифметичних дій над числами в десятковій системі числення. Автором підручника був арабський математик Мухаммед Бен Муса аль-Хорезмі. Від слова „альджебр" у назві підручника пішло слово „алгебра", а від імені аль-Хорезмі – слово „алгоризм", що пізніше перейшло в слово „алгоритм".

Властивості алгоритмів :

1. *Зрозумілість*. В алгоритмі повинні бути лише операції, які знайомі виконавцеві. При цьому виконавцем алгоритму може бути: людина, комп'ютер, робот тощо.
2. *Масовість*. За допомогою складеного алгоритму повинен розв'язуватися цілий клас задач.
3. *Однозначність*. Будь-який алгоритм повинен бути описаний так, щоб при його виконанні у виконавця не виникало двозначних вказівок. Тобто різні виконавці згідно з алгоритмом повинні діяти однаково та прийти до одного й того ж результату.
4. *Правильність*. Виконання алгоритму повинно давати правильні результати.
5. *Скінченність*. Завершення роботи алгоритму повинно здійснюється в цілому за скінченну кількість кроків.
6. *Дискретність*. Алгоритм повинен складатися з окремих завершених операцій, які виконуються послідовно.
7. *Ефективність*. Алгоритм повинен забезпечувати розв'язання задачі за мінімальний час з мінімальними витратами оперативної пам'яті.

Способи представлення алгоритмів. Алгоритми можуть бути представлені: у вигляді таблиці, описані як система словесних правил (лексикографічний або словеснокроковий спосіб запису алгоритму), представлені алгоритмічною мовою у вигляді послідовності операторів

(операторний спосіб), або з допомогою графічного зображення у формі блок-схем (графічний або геометричний спосіб запису алгоритму).

Слід зауважити, що графічному способу подання алгоритмів надається перевага через його простоту, наочність і зручність. *Блок-схема* алгоритму зображає послідовність блоків, з'єднаних між собою стрілками, які вказують послідовність виконання і зв'язок між блоками. Всередині блоків записується їх короткий зміст.

1.5.2 Блок-схеми

Блок-схема - це спосіб представлення алгоритму в графічній формі, у вигляді геометричних фігур, сполучених між собою лініями (стрілками). Форма блока визначає тип дії, а текст всередині блоку дає детальне пояснення конкретної дії. Стрілки на лініях, що сполучають блоки схеми, вказують послідовність виконання команд, передбачених алгоритмом. Блок-схеми, за рахунок наочності спрощують створення ефективних алгоритмів, розуміння роботи вже створених, а як наслідок і їх оптимізацію. Існуючі стандарти на типи блоків дозволяють легко адаптувати алгоритми, створені у вигляді блок-схем до будь-яких існуючих на сьогоднішній день мов програмування.

Зображення блоків у алгоритмі, їх розміри, товщина ліній, кут нахилу ліній тощо, регламентуються Державним стандартом "Схеми алгоритмів, програм, даних і систем", а саме : 19.701-90 (ISO 5807-85).

Блоки у блок-схемі з'єднуються лініями потоків. У кожен блок може входити не менше однієї лінії, з блоку ж (окрім логічного) може виходити лише одна лінія потоку . З логічного блоку завжди виходять дві лінії потоку: одна у випадку виконання умови, інша - при її невиконанні. Бажано, щоб лінії потоку не перетинались.

Алгоритм може бути детальним, або спрощеним (деякі зрозумілі блоки можуть не записуватись, інакше алгоритм збільшується в розмірі).

Основні види блок-схем :

- прості (нерозгалужені);
- розгалужені;
- циклічні;
- з підпрограмами;
- змішані.

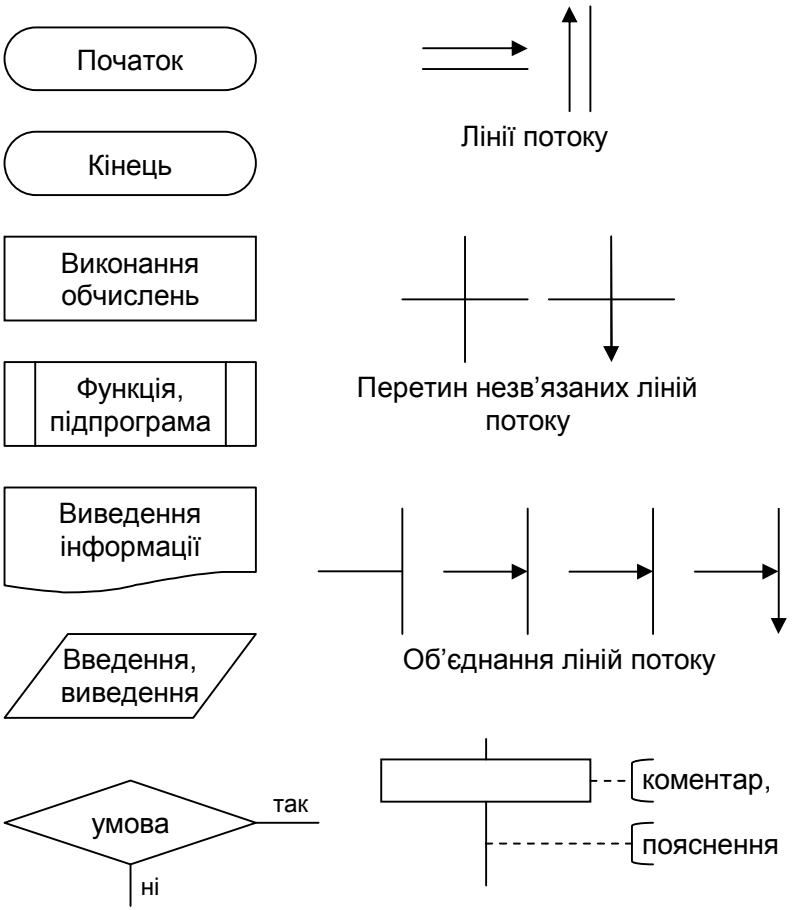


Рис. 1.2. Основні елементи блок-схем

1.5.3 Базові алгоритмічні конструкції:

Базові алгоритмічні конструкції - це способи управління процесами обробки даних. Виділяють три базові алгоритмічні конструкції:

1. лінійні алгоритми ;
2. алгоритми розгалуженої структури;
3. алгоритми циклічної структури.

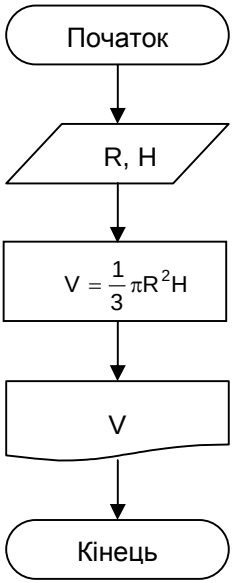


Рис.1.3. Приклад лінійного алгоритму

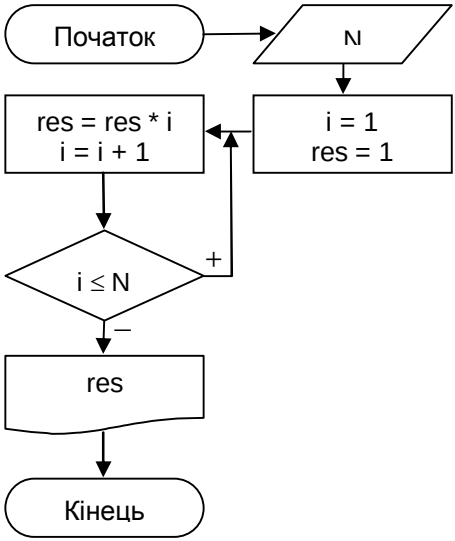


Рис.1.4. Приклад розгалуженого алгоритму

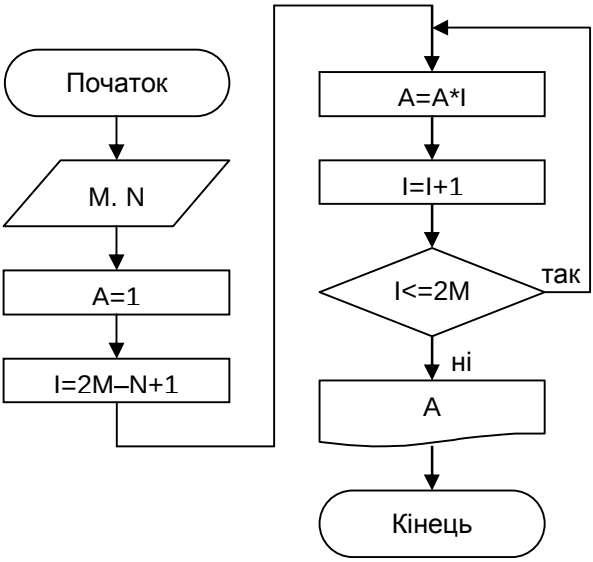


Рис.1.5. Приклад циклічного алгоритму

Лінійні алгоритми (рис. 1.3). Алгоритм називається *лінійним*, якщо блоки алгоритму виконуються один за одним. Алгоритми лінійної структури не містять умовних і безумовних переходів, циклів.

Алгоритми розгалуженої структури (рис.1.4). Якщо вибраний метод розв'язання задачі передбачає виконання різних дій в залежності від значень будь-яких змінних, але при цьому кожна гілка алгоритму в процесі розв'язання задачі виконується не більше одного разу, алгоритм називається *розгалуженим*.

Алгоритми циклічної структури (рис.1.5).

Цикл - це команда виконавцеві (компілятору) багаторазово повторити послідовність певних команд.

При багатократному проходженні деяких ділянок алгоритму в процесі виконання алгоритм називається *циклічним*. Кількість проходжень циклу повинна бути повністю визначена алгоритмом розв'язання задачі, інакше виникає "зациклювання", при якому процес розв'язання задачі не може завершитися.

Алгоритми розв'язку задач циклічної структури можуть бути такими, що при однократному проході циклу деякі ділянки алгоритму виконуються неодноразово, тобто всередині циклу існують інші цикли. Алгоритми такої структури називаються *алгоритмами з вкладеними циклами*.

1.6 Оператори

Тепер перейдемо до запису алгоритмів програм безпосередньо мовою програмування Сі.

Оператори – це основні елементи, з яких „будуються” програми на будь-якій мові програмування. Більшість операторів складаються з виразів. Виходячи з цього, спочатку розглянемо вирази.

Вираз представляє собою об’єднання операцій і операндів. Найпростіший вираз складається з одного операнду.

Приклади виразів :

```
5
-7
10+21
a*(b+d*1)-1
x=++a%3
a>3
```

Неважко помітити, що операнди можуть бути константами, змінними, їх об'єднаннями. Деякі вирази складаються з менших виразів.

Дуже важливою особливістю мови Сі є те, що кожний вираз має значення. Наведемо приклади кількох виразів і їх значень :

–5+7	2
1<2	1
6+(a=1+2)	9
a=1+2	3

Як вже було сказано, основу будь-якої програми складають оператори. Оператором-виразом називається вираз, вслід за яким стоїть крапка з комою. Взагалі усі оператори можна згрупувати у наступні класи:

- оператори присвоювання;
- виклики функцій;
- розгалуження;
- цикли.

Проте, оператори найчастіше відносяться до більш ніж одного з чотирьох класів. Наприклад, оператор if *(a=fn(b+c)>d)* складається з представників наступних класів : присвоювання, виклик функції та розгалуження. У тому і є гнучкість Сі, що є можливість змішування в одному операторі операторів різних класів. Проте навряд чи слід цим зловживати – програма може вийти правильною, проте надто заплутаною та нечитабельною.

1.6.1 Оператор розгалуження if

Оператор *розгалуження* призначений для виконання тих або інших дій в залежності від істинності або хибності деякої умови. Основний оператор цього блоку в *Ci – if … else* не має ключового слова *then*, як у Паскалі, проте обов'язково вимагає, щоб умова, що перевіряється, розміщувалася б у круглих дужках. Оператор, що слідує за логічним виразом, є *then*- частиною оператору *if...else***.**

Синтаксис оператора :

```
if (<умова>)
        <оператор1>;
   [else <оператор2;>]
```

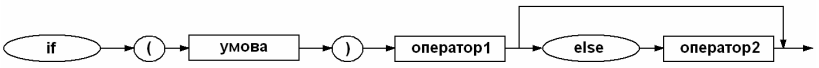


Рис. 1.6. Синтаксис оператора if

Умова хибна, якщо вона дорівнює нулю, в інших випадках вона істинна. Це означає, що навіть від'ємні значення розглядаються як істинні. До того ж, умова, що перевіряється, повинна бути скалярною, тобто зводитися до простого значення, яке можливо перевірити на рівність нулю. Взагалі не рекомендується використання змінних типу *float* або *double* в логічних виразах перевірки умов з причини недостатньої точності подібних виразів. Більш досвідчені програмісти скорочують оператори типу:

```
if (вираз!=0) оператор;
```

до наступного:

```
if (вираз) оператор;.
```

Обидва логічні вирази функціонально еквівалентні, тому що *будь-яке ненульове значення розцінюється як істина*. Це можна довести наступними програмами:

Приклад 1.

```
/* програма виводить результат ділення двох дійсних
чисел */
#include<stdio.h>
#include<conio.h>
void main()
{
  float a,b,c;
  printf("Введіть число a :\n");
  scanf("%f",&a);
  printf("Введіть число b :\n");
  scanf("%f",&b);
  if (b==0) printf("Ділення да нуль !\n");
    else
    {
      c=a/b;
      printf("a : b == %g",c);
    };
}
```

Приклад 2.

```
/* застосування умовного розгалужування */
#include <stdio.h>
main()
{
  int number;
  int ok;
  printf("Введіть число з інтервалу 1..100 : ");
  scanf("%d",&number);
  ok=(1<=number) && (number<=100);
  if (!ok)
  printf("Не коректно !!\n");
  return ok;
}
```

Змінній *ok* присвоюється значення результату виразу: ненульове значення, якщо істина, і в протилежному випадку - нуль. Умовний оператор *if(!ok)* перевіряє, якщо *ok* дорівнюватиме нулю, то *!ok* дасть позитивний результат й відтоді буде отримано повідомлення про некоректність, виходячи з контексту наведеного прикладу.

1.6.2 Оператор switch

Синтаксис :

```
switch(<вираз цілого типу>)
{
  case <значення_1>:
        <послідовність_операторів_1>;
        break;
  case <значення_2>:
        <послідовність_операторів_2>;
        break;
  .....
  case <значення_n>:
        <послідовність_операторів_n>;
        break;
  [default:
        <послідовність_операторів_n+1>;]
}
```

Оператор-перемикач *switch* призначений для вибору одного з декількох альтернативних шляхів виконання програми. Виконання оператора *switch* починається з обчислення значення виразу (виразу, що слідує за ключовим словом *switch* у круглих дужках). Після цього

управління передається одному з <*операторів*>. Оператор, що отримав управління – це той оператор, значення константи варіанту якого співпадає зі значенням виразу перемикача.

Вітка *default* (може опускатися, про що свідчить наявність квадратних дужок) означає, що якщо жодна з вищенаведених умов не задовольнятиметься (тобто вираз цілого типу не дорівнює жодному із значень, що позначені у *case*-фрагментах), керування передається по замовчуванню в це місце програми. Треба також зазначити обов'язкове застосування оператора *break* у кожному з *case*-фрагментів (цей оператор застосовують для негайного припинення виконання операторів *while*, *do*, *for*, *switch*), що негайно передасть керування у точку програми, що слідує відразу за останнім оператором у *switch*-блоці.

Приклад 1:

```
switch(i)
{
 case –1:
   n++;
   break;
 case 0:
   z++;
   break;
 case 1:
   p++;
   break;
}
```

Приклад 2 :

```
switch(c)
{
 case 'A':
   capa++;
 case 'a':
   lettera++;
 default:
   total++;
}
```

В *останньому прикладі* всі три оператори в тілі оператора *switch* будуть виконані, якщо значення *c* рівне 'A', далі оператори виконуються в порядку їх слідування в тілі, так як відсутні *break*.

1.6.3 Оператор циклу з передумовою while

Оператор *while* використовується для організації циклічного виконання оператора або серії операторів, поки виконується певна умова.

Синтаксис :

while (<логічний вираз>)
  оператор;

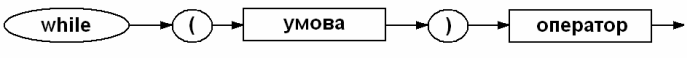


Рис. 1.7. Синтаксис оператора while

Цикл закінчується у наступних випадках :

1. умовний вираз у заголовку приймає нульове значення;
2. у тілі циклу досягнуто місця, де розташований оператор *break*;
3. у тілі циклу виконаний оператор *return*;

У перших двох випадках керування передається оператору, розташованому безпосередньо за циклом, у третьому випадку активна на той момент функція завершує свою роботу, повертаючи якесь значення.

Знову ж таки нерідкою помилкою програмістів, що працювали раніше на Паскалі, є використання замість оператора порівняння *(==)* оператора присвоювання *(=)*. Наприклад, наступний фрагмент створить нескінчений цикл:

```
/* некоректне  використання оператору циклу */
int main(void)
{
  int j=5;
  while(j=5)    /* змінній j присвоїти значення 5 */
  {
     printf("%d\n",j);
     j++;
  }
}
```

Компілятор *Ci* попередить про некоректне присвоювання в даному випадку, виправити яке особливих труднощів не викличе.

Втім, часто такий цикл використовується для перевірки відповіді користувача на питання з програми ("так чи ні ?"):

```
/* фрагмент використання while */
printf ("Підтверджуєте ? Так чи ні ?(y/n);");
scanf("%c",&ch);
while (ch!='y' && ch!='n')
{
  printf("\n Відповідайте так чи ні . . (y/n);");
  scanf("%c",&ch);
}
```

Тіло циклу почне виконуватися, якщо користувач введе будь-який символ, відмінний від *y* або *n*. Цикл виконується доти, доки користувач не введе або '*y'* , або '*n'*.

Цікаво розглянути й наступний приклад, що застосовує оператор *while* у функції підрахунку факторіалу:

```
long factorial(int number)
{
  long total;
  total=number;
  while (--number)
     total*=number;
  return total;
}
```

1.6.4 Оператор циклу з постумовою do … while

Оператор *do…while* використовується для організації циклічного виконання оператора або серії операторів, які називаються тілом циклу, до тих пір, поки умова не стане хибною.

Синтаксис :

do
    <оператор>;
while (<логічний_вираз>);

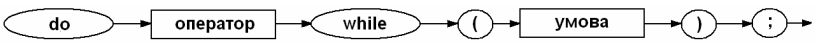


Рис. 1.8. Синтаксис оператора do … while

Ситуації, що призводять до виходу з циклу, аналогічні наведеним для циклу *while* із передумовою. Характерним є те, що тіло циклу виконається хоча б один раз. На відміну від Паскаля, в якому цикл з постумовою *repeat operator until умова* виконується, поки умова невірна, цикл *do ... while* навпаки припиняє виконання, коли умовний вираз обертається в нуль (стає невірним).

Приклад 1.

```
printf ("Підтверджуєте ? Так чи ні ?(y/n);");
do
  scanf("%c",&ch);
while (ch!='y' && ch!='n')
```

Приклад 2.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int n,i;
  float fact;
  printf("Програма обчислення n!.\n");
  printf("Введіть число n :\n");
  scanf("%d",&n);
  i = 1;
  fact = 1;
  do {
    fact *= i;
    i++;
  }
    while (i <= n);
  printf("n!==%g",fact);
}
```

1.6.5 Оператор розриву break

Синтаксис :

break;

Оператор розриву *break* перериває виконання операторів *do*, *for*, *while* або *switch*.

В операторі *switch* він використовується для завершення блоку *case*.

В операторах циклу – для негайного завершення циклу, що не зв'язане з перевіркою звичайної умови завершення циклу. Коли оператор *break* зустрічається всередині оператора циклу, то здійснюється негайний вихід з циклу і перехід до виконання оператору, що слідує за оператором циклу.

Приклад :

```
main()
{
   int i;
   for (i=0;i<1000;i++)
   {
      printf("%d - %d\n",i,i*i*i);
      if (i*i*i>=10000) break;
   }
   return 0;
}
```

1.6.6 Оператор продовження continue

Синтаксис :

continue;

Оператор *continue* передає управління на наступну ітерацію в операторах циклу *do*, *for*, *while*. Він може розміщуватися тільки в тілі цих операторів. В операторах *do* і *while* наступна ітерація починається з обчислення виразу умови. Для оператора *for* наступна ітерація починається з обчислення виразу зміни значення лічильника.

Приклад :

```
while (i-- > 0)
{
  x=f(i);
  if (x == 1) continue;
     else y=x*x;
}
```

В даному прикладі тіло циклу *while* виконується якщо *i* більше нуля. Спочатку значення *f(i)* присвоюється змінній *x*;потім, якщо *x* не рівний 1, то *y* присвоюється значення квадрата числа *x*, і управління передається на „заголовок" циклу, тобто на обчислення виразу (i-- > 0). Якщо ж *x* рівний 1, то виконується оператор продовження *continue*, і виконання продовжується з „заголовку" оператора циклу *while*, без обчислення квадрата *x*.

1.6.7 Оператор циклу for

Оператор *for* забезпечує циклічне повторення деякого оператора певне число разів. Оператор, який повторюється називається тілом циклу. Повторення циклу звичайно здійснюється з використанням деякої змінної (лічильника), яка змінюється при кожному виконанні

тіла циклу. Повторення завершується, коли лічильник досягає заданого значення.

Синтаксис оператора:

```
for([ініціалізація];[перевірка_умови];[нове_значення])
    оператор ;
```

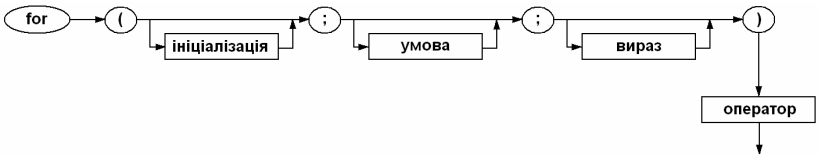


Рис. 1.9. Синтаксис оператора for

Звернемо увагу на те, що кожен з трьох виразів може бути відсутнім. Перший вираз служить для ініціалізації лічильника, другий - для перевірки кінця циклу, а третій вираз - для зміни значення лічильника. Формально роботу циклу можна описати такими кроками:

1. якщо перший вираз (ініціалізація) присутній, то він обчислюється;
2. обчислюється вираз умови (якщо він присутній). Якщо умова виробляє значення 0, тобто вона невірна, цикл припиняється, у протилежному випадку він буде продовжений;
3. виконується тіло циклу;
4. якщо присутній вираз зміни лічильника, то він обчислюється;
5. надалі перехід до пункту під номером *2*.

Поява у будь-якому місці циклу оператора *continue* призведе до негайного переходу до пункту 4.

Приклад використання циклу *for* :

```
/* друк парних чисел у проміжку від 500 до 0  */
#include <stdio.h>
void main(void) {
  long i;
  for(i=500;i>=0;i-=2)
     printf("\n%ld",i);
  printf("\n");
}
```

Для того, щоб продемонструвати гнучкість даного різновиду циклу, розглянемо інші варіанти цієї ж програми. У першому випадку

представимо весь перелік обчислень лише в одному операторі *for*, за яким слідує порожній оператор:

```
#include <stdio.h>
int main(void)
{
  long i;
  for(i=500;i>=0;printf("\n%ld",i),i-=2) ;
}
```

Другий варіант використовує оператор *continue*:

```
#include <stdio.h>
int main(void)
{
 long i;
 for(i=500;i>=0;i--)
    if (i%2 == 1)
     continue;
   else
     printf("\n %ld", i );
     printf("\n");
}
```

Справа програміста, який з варіантів обрати - надати перевагу більш стислому викладанню або навіть взагалі скористатися іншим оператором. Цікаво, що різновид циклу *for* можна звести до циклу *while* наступним чином:

```
for(вираз1;вираз2;вираз3)
   оператор;

/* далі – аналогічний цикл while  */
вираз1;
while (вираз2)
{
  оператор;
  вираз3;
}
```

Інша справа - чи є в такій заміні необхідність? Не завжди гнучкість переважає стисліcть та навпаки. Справа за конкретною ситуацією. Зрештою, вибір циклу може бути й справою смаку конкретного програміста – саме йому вирішувати, які оператори застосувати для вірного запису того чи іншого алгоритму.

1.6.8 Оператор переходу goto

Синтаксис :

goto <мітка>;

…

<мітка> : <оператор>;

Оператор безумовного переходу *goto* передає управління безпосередньо на <оператор>, перед яким розташована <мітка>. Область дії мітки обмежена функцією, в якій вона визначена. Тому, кожна мітка повинна бути відмінною від інших в одній і тій самій функції. Також, неможливо передати управління оператором *goto* в іншу функцію.

Оператор, перед яким розташована <мітка> виконується зразу після виконання оператора *goto*.

Якщо оператор з міткою відсутній, то компілятор видасть повідомлення про помилку.

Приклад використання *goto*:

```
if (errorcode>0)
   goto exit;
…
exit :
return errorcode;
```

В свою чергу при появі концепції структурного програмування оператор *goto* піддався критиці, і його використання стало розглядатися як ознака поганого стилю програмування. Дійсно, надмірно широке використання *goto* робить структуру програми надмірно заплутаною, тому без особливої необхідності намагайтесь обходитися без оператора *goto*.

1.6.9 „Порожній" оператор

Синтаксис : ;

Порожній оператор – це оператор що складається лише з крапки з комою. Він може використовуватися в будь-якому місці програми, де за синтаксисом потрібний оператор.

```
for (i=0;i<10;printf("%d\n",i);) ;
```

1.6.10 „Складений" оператор

„Складений" оператор представляє собою два або більше операторів. Його також називають „блоком".

Синтаксис :

```
{
   [<оператори>]
}
```

Дія складеного оператора полягає в обов'язковому послідовному виконанні операторів, які містяться між { та }, за виключенням тих випадків, коли який-небудь оператор явно не передасть управління в інше місце програми.

```
if (i>0)
{
  printf("i == %d\n",i);
  i--;
}
```

1.7 Тип перерахування enum

При написанні програм часто виникає необхідність визначити декілька іменованих констант, для яких потрібно, щоб всі вони мали різні значення (при цьому конкретні значення можуть бути не важливими). Для цього зручно скористатися типом даних „перерахування" *enum* (enumeration), всі можливі значення якого задаються списком цілочисельних констант.

Синтаксис :

enum [ім'я_типу] { список_констант };

Ім'я типу задається тоді, коли в програмі є необхідність визначати змінні даного типу. Компілятор забезпечує, щоб ці змінні приймали значення тільки із вказаного списку констант.

```
enum {mRead, mEdit, mWrite, mCreate } Mode;
```

Цей оператор вводить іменовані константи *mRead*, *mEdit*, *mWrite* і змінну *Mode*, яка може приймати значення цих констант. В момент оголошення змінна ініціалізується значенням першої константи, в наведеному прикладі – *mRead*. В подальшому їй можна присвоювати будь-які допустимі значення. Наприклад :

```
Mode = mCreate;
```

Значення змінної типу перерахування можна перевіряти, порівнюючи її з можливими значеннями. Крім того, потрібно враховувати, що типи перерахування відносяться до цілих порядкових типів і до них можуть бути застосовані будь-які операції порівняння. Наприклад :

```
if (Mode>mRead) /* … */ ;
```

Змінну *Mode* можна також використовувати в структурі *switch*:

```
switch(Mode)
{
  case mRead:    /* … */
       break;
  case mEdit:    /* … */
       break;
  case mWrite:   /* … */
       break;
  case mCreate: /* … */
       break;
}
```

По замовчуванню значення, які вказані в *enum*, інтерпретуються як цілі числа, причому перше значення рівне 0, друге – 1 і т.д. Значення по замовчанню можна змінити, якщо після імені константи вказати знак рівності і задати ціле значення константи. Наприклад :

```
enum {mRead = –1, mEdit, mWrite = 2, mCreate } Mode;
```

Якщо після констант не задане їх ціле значення, воно вважається на 1 більшим, ніж попереднє. Тому для нашого прикладу значення констант такі:

mRead	–1
mEdit	0
mWrite	2
mCreate	3

1.8 Покажчики

1.8.1 Основні відомості про покажчики

В результаті процесу компіляції програми всі імена змінних будуть перетворені в адреси комірок пам'яті, в яких містяться відповідні значення даних. У командах машинної програми при цьому знаходяться машинні адреси розміщення значень змінних. Саме це і є

пряма адресація – виклик значення за адресою в команді. Наприклад, в операторі присвоювання: *k = j* на машинному рівні відбувається копіювання значення з області ОП, що відведена змінній *j*, в область ОП, яка відведена змінній *k*. Таким чином, при виконанні машинної програми реалізуються операції над операндами – значеннями змінних, розташованими за визначеними адресами ОП. На машинному рівні імена змінних у командах не використовуються, а тільки адреси, сформовані транслятором з використанням імен змінних. Проте програміст не має доступу до цих адрес, якщо він не використовує покажчики.

Покажчики в *Ci* використовується набагато інтенсивніше, аніж, скажімо, у Паскалі, тому що іноді деякі обчислення виразити можливо лише за їх допомогою, а частково й тому, що з ними утворюються більш компактні та ефективніші програми, аніж ми використовували б звичайні засоби. Навіть існує твердження - аби стати знавцем *Ci*, потрібно бути спеціалістом з використання покажчиків.

Покажчик (вказівник) - це змінна або константа стандартного типу даних для збереження адреси змінної визначеного типу. Значення покажчика - це беззнакове ціле, воно повідомляє, де розміщена змінна, і нічого не говорить про саму змінну.

Тип змінної, що адресується, може бути стандартний, нумерований, структурний, об'єднання або *void*. Покажчик на тип *void* може адресувати значення будь-якого типу. Розмір пам'яті для самого покажчика і формат збереженої адреси (вмісту покажчика) залежить від типу комп'ютера та обраної моделі пам'яті. Константа ***NULL*** зі стандартного файлу *stdio.h* призначена для ініціалізації покажчиків нульовим (незайнятим) значенням адреси.

Змінна типу покажчик оголошується подібно звичайним змінним із застосуванням унарного символу “*“. Форма оголошення змінної типу покажчик наступна:

тип [модифікатор] * імені-покажчика ;

де тип – найменування типу змінної, адресу якої буде містити змінна-покажчик (на яку він буде вказувати).

Модифікатор необов'язковий і може мати значення:

- *near* - ближній, 16-бітний покажчик (встановлюється за замовчуванням), призначений для адресації 64-кілобайтного сегмента ОП;
- *far* - дальній, 32-бітний покажчик, містить адресу сегмента і зсув у ньому: може адресувати ОП обсягом до 1 Мб;

- *huge* - величезний, аналогічний покажчику типу *far*, але зберігається у нормалізованому форматі, що гарантує коректне виконання над ним операцій; застосовується до функцій і до покажчиків для специфікації того, що адреса функції або змінної, що адресується, має тип *huge*;
- імені-покажчика - ідентифікатор змінної типу покажчик;
- визначає змінну типу покажчик.

Значення змінної-покажчика - це адреса деякої величини, ціле без знака. Покажчик містить адресу першого байту змінної визначеного типу. Тип змінної, що адресується, і на яку посилається покажчик, визначає об‘єм ОП, що виділяється змінній, та зв'язаному з нею покажчикові. Для того, щоб машинною програмою обробити (наприклад, прочитати або записати) значення змінної за допомогою покажчика, треба знати адресу її початкового (нульового) байта та кількість байтів, що займає ця змінна. Покажчик містить адресу нульового байту цієї змінної, а тип змінної, що адресується, визначає, скільки байтів, починаючи з адреси, визначеної покажчиком, займає це значення.

Нижче наведено приклади деяких можливих оголошень покажчиків:

```
int *pi;  /* - покажчик - змінна на дані типу int */
float *pf; /* - покажчик - змінна на дані типу float */
int ml [5]; /* - ім'я масиву  на 5 значень типу int;
   ml - покажчик-константа, про це йтиметься згодом */
int *m2[10]; /* m2 - ім'я масиву на 10 значень типу
     покажчик на  значення   типу int, m2 - покажчик-
       константа */
int (*m3)[10]; /* - покажчик на масив з 10 елементів
            типу int; m3 - покажчик-константа */
```

Зверніть увагу на те, що у трьох з наведених оголошень ім’я масиву є константою - покажчиком! (Про це йтиметься в наступному окремому розділі.)

За допомогою покажчиків, наприклад, можна:

1. обробляти одновимірні та багатовимірні масиви, рядки, символи, структури і масиви структур;
2. динамічно створювати нові змінні в процесі виконання програми;
3. обробляти зв'язані структури: стеки, черги, списки, дерева, мережі;
4. передавати функціям адреси фактичних параметрів;
5. передавати функціям адреси функцій в якості параметрів.

Протягом довгого часу програмісти були незадоволені покажчиками. Зокрема, застосування покажчиків критикується через те, що в силу їх природи неможливо визначити, на яку змінну вказує в даний момент покажчик, якщо не повертатися до того місця, де покажчику востаннє було присвоєно значення. Це ускладнює програму і робить доведення її правильності дещо ускладненим. Програміст, що добре володіє Сі, повинен насамперед знати, що таке покажчики, та вміти їх використовувати. Практично у програмі можна використовувати не імена змінних, а тільки покажчики, тобто адреси розміщення змінних програми.

1.8.2 Моделі пам'яті

У мові Сі для операційної системи MS-DOS розмір ОП (оперативної пам’яті) для розміщення покажчика залежить від типу використаної моделі пам'яті. У програмах на мові Сі можна використовувати одну з шести моделей пам'яті: крихітну (*tiny*), малу (*small*, по замовчуванню), середню (*medium*), компактну (*compact*), велику (*large*) і величезну (*huge*).

Взагалі оперативна пам'ять для виконання програми на мові Сі використовується для:

- розміщення програми (коду програми);
- розміщення зовнішніх (глобальних) і статичних даних (що мають специфікатори *extern* і *static*, про них йтиметься нижче);
- динамічного використання ОП для змінних, сформованих у процесі виконання програми (купа, динамічна ОП, про них йтиметься нижче);
- для розміщення локальних (*auto* - автоматичних) змінних, змінних функцій (стек) під час виконання програми.

Старші адреси молодші адреси

Буфери, ПЗП, відео-пам'ять	Невикористана пам'ять	Стек	Вільна ОП	Купа	Статич. Дані	Код прогр.	Вектори переривання DOS

де ПЗП - постійний запам'ятовуючий пристрій

Рис. 1.10. Структура оперативної пам'яті

ОП програми та її статичних даних у процесі виконання програми залишається незмінною. ОП з купи виділяється та звільняється в

процесі виконання програми. Об‘єм ОП для купи залежить від того, скільки ОП запитує програма за допомогою функцій *calloc()* та *malloc()* для динамічного розміщення даних. Пам'ять стека виділяється для фактичних параметрів активізованих функцій і їх локальних (автоматичних) змінних. Розглянемо основні характеристики різних моделей ОП.

Крихітна (tiny model) ОП. Модель пам'яті використовується при дефіциті ОП. Для коду програми, статичних даних, динамічних даних (купи) та стеку виділяється 64 Кб. Змінна - покажчик типу *near* (ближній) займає 2 байти.

Мала (small model) ОП. Для програми призначається 64 Кб. Стек, купа і статичні дані займають по 64 Кб. Ця модель приймається по замовчуванню та використовується для вирішення маленьких і середніх задач. Покажчик типу *near* займає 2 байти і містить адресу - зсув усередині сегмента ОП з 64 Кб.

Середня (medium model) ОП. Розмір ОП для програми дорівнює 1 Мбайт. Стек, купа і статичні дані розміщаються в сегментах ОП розміром 64 Кб. Цю модель застосовують для дуже великих програм і невеликих обсягів даних. Покажчик у програмі типу *far* займає 4 байти. Для адресації даних покажчик типу *near* займає 2 байти.

Компактна (compact model) ОП. Для програми призначається 64 Кб. Для даних - 1 Мбайт. Об‘єм статичних даних обмежується 64 Кб. Розмір стека повинен бути не більш 64 Кб. Ця модель використовується для малих і середніх програм, що вимагають великого об‘єму даних. Покажчики в програмі складаються з 2 байтів, а для даних - з 4 байтів.

Велика (large model) ОП. ОП для програми обмежена 1 Мб. Для статичних даних призначається 64 Кб. Купа може займати до 1 Мб. Програма і дані адресуються покажчиками, що займають 4 байти. Модель використовується для великих задач. Окрема одиниця даних, наприклад масив, повинна займати не більш 64 Кб.

Величезна (huge model) ОП. Аналогічна великій моделі. Додатково в ній знімається обмеження на розмір окремої одиниці даних.

1.8.3 Основні операції над покажчиками

Мова Сі надає можливість використання адрес змінних програми за допомогою основних операцій - & та *:

За допомогою основних операцій можна отримати значення адреси змінної а використовуючи непряму адресацію - одержати значення змінної за її адресою.

Призначення цих операцій:

& ім‘я змінної – одержання адреси; визначає адресу розміщення значення змінної визначеного типу;

* ім‘я-покажчика – отримання значення визначеного типу за вказаною адресою; визначає вміст змінної, розміщеної за адресою, що міститься у даному покажчику; це - непряма адресація (інші назви - “зняття значення за покажчиком” або “розіменування”).

Оператор присвоювання значення адреси покажчику має вигляд:

Ім‘я_змінної_покажчика = & ім‘я змінної;

Наприклад:

```
int i, *pi;    /* pi –змінна покажчик   */
pi = &i;       /* pi одержує значення адреси 'i' */
```

Операція & - визначення адреси змінної повертає адресу ОП свого операнда. Операндом операції & повинне бути ім'я змінної того ж типу, для якого визначений покажчик лівої частини оператора присвоювання, що одержує значення цієї адреси. У вищенаведеному прикладі це тип *int*.

Операції * і & можна писати впритул до імені операнду або через пробіл. Наприклад: &i, * pi.

Непряма адресація змінної за допомогою операції * здійснює доступ до змінної за покажчиком, тобто повернення значення змінної, розташованої за адресою, що міститься у покажчику. Операнд операції * обов’язково повинен бути типу покажчик. Результат операції * - це значення, на яке вказує (адресує, посилається) операнд. Тип результату - це тип, визначений при оголошенні покажчика.

У загальному вигляді оператор присвоювання, що використовує ім'я покажчика та операцію непрямої адресації, можна представити у вигляді:

ім‘я змінної * ім‘я-покажчика;

де ім‘я-покажчика - це змінна або константа, що містить адресу розміщення значення, необхідного для змінної лівої частини оператора присвоювання.

Наприклад:

```
i= *pi; /* 'i' одержує значення, розташоване за
           адресою, що міститься в покажчику 'pi' */
```

Як і будь-які змінні, змінна pi типу покажчик має адресу і значення. Операція & над змінною типу покажчик: &pi – дає адресу місця розташування самого покажчика, pi – ім'я покажчика визначає його значення, а *pi – значення змінної, що адресує покажчик.

Звичайно, усі ці значення можна надрукувати. Наприклад, за допомогою наступної програми:

```
#include <stdio.h>
void main()
{
  char c = 'A';
  int i = 7776;
  int *pi = &i;
  char *pc = &c;
  printf ("pi=%u,*pi=%d, &pi=%u\n", pi, *pi, &pi);
  printf ("pc=%u, *pc=%c, &pc=%u\n", pc, *pc, &pc);
}
```

У результаті виконання буде виведено:

```
pi = 65522, *pi = 7776, &pi = 65520
pc = 65525, *pc = A, &pc = 65518
```

Одне з основних співвідношень при роботі з покажчиками - це симетричність операцій адресації та непрямої адресації. Вона полягає в тому, що:

&x == x, тобто вміст за адресою змінної x є значення x.

Наприклад, оголошення покажчика pi і змінних i та j:

```
int *pi, i = 123, j;
pi = &i; /*-присвоювання покажчику значення адреси i */
j = *pi; /* - присвоювання j вмісту за адресою pi */
```

Тут змінна *j* отримує вміст, розташований за адресою змінної i, тобто значення змінної, що адресує покажчик *pi*: *j = * pi = * &i = i;*. Два останніх вищенаведених оператора виконують те саме, що один оператор: *j = i*.

Для повного остаточного розуміння процесів, що відбувається у пам'яті при маніпуляції з покажчиками, розглянемо ще такий фрагмент:

```
void func() {
  int x;
  int *px; /* px - покажчик на змінну типу int*/
  px= &x ; /* адреса змінної x заноситься в px*/
  *px=77;  /* число зберігається за адресою, на яку
              вказує px */
}
```

Розглянемо цей приклад на конкретному малюнку: функція займає область пам'яті, починаючи з адреси *0x100*, *x* знаходиться за адресою *0x102*, а *px* - *0x106*. Тоді перша операція присвоювання, коли значення *&x(0x102*) зберігається в *px,* матиме вигляд, зображений на рис. 1.11 зліва:

Наступну операцію, коли число 77 записується за адресою, яка знаходиться в *px* та дорівнює *0x102* (адреса x), відображає рис. 1.11 справа. Запис **px* надає доступ до вмісту комірки, на яку вказує *px*.

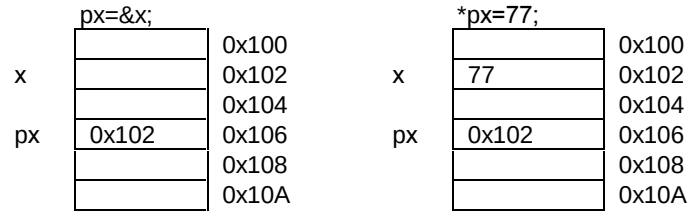


Рис. 1.11. Схематичне представлення значень в ОП

Далі наведений приклад програми виводу значень покажчика і вмісту, розташованого за адресою, що він зберігає.

```
#include<stdio.h>
void main()
{
  int i = 123, *pi = &i; /* pi-покажчик на значення
типу int */
  printf("розмір покажчика pi = %d\n", sizeof(pi));
  printf("адреса розміщення покажчика pi=%u\n", &pi) ;
  printf("адреса змінної i = %u\n", &i) ;
  printf("значення покажчика pi = %u\n", pi) ;
  printf("значення за адресою pi = %d\n", *pi) ;
  printf("значення змінної i = %d\n", i) ;
}
```

Результати виконання програми:

```
розмір покажчика pi = 2
адреса розміщення покажчика pi = 65522
адреса змінної i= 65524
значення покажчика pi = 65524
значення за адресою pi = 123
значення змінної i = 123
```

Покажчики можна використовувати:

1. у виразах, наприклад, для одержання значень, розташованих за адресою, що зберігається у покажчику;
2. у лівій частині операторів присвоювання, наприклад:
   a. для одержання значення адреси, за якою розташоване значення змінної;
   b. для одержання значення змінної.

Наприклад, якщо *pi* - покажчик цілого значення (змінної *i*), то **pi* можна використовувати в будь-якому місці програми, де можна використовувати значення цілого типу. Наприклад:

```
int i = 123, j, *pi;
pi = &i; /*pi у лівій частині оператора присвоювання */
j = *pi + 1; /*-це еквівалентно: j = i + 1;
    pi-у виразі правої частини оператора присвоювання*/
```

Виклик значення за покажчиком можна використовувати також як фактичні параметри при звертанні до функцій. Наприклад:

```
d = sqrt ((double) *pi); /* *pi - фактичний параметр */
fscant (f, "%d", pi );   /*  pi - фактичний параметр */
printf ("%d\n", *pi );   /* *pi - фактичний параметр */
```

У виразах унарні операції & і *, пов'язані з покажчиками, мають більший пріоритет, ніж арифметичні. Наприклад:

```
*px = &x;
y = 1 + *px; /*-спочатку виконується '*', потім '+' */
```

Останній оператор еквівалентний:

```
y = 1 + x;
```

Для звертання до значення за допомогою покажчика-змінної його можна використовувати в операторі присвоювання скрізь, де може бути ім'я змінної. Наприклад, після виконання оператора: *px = &x;* цілком еквівалентними є такі описи:

Оператор:	Його еквівалент:	Або:
`*px =0;`	`x = 0;`	
`*px += 1;`	`*px = *px + 1;`	`x = x + 1;`
`(*px)++ ;`	`*px = *px + 1;`	`x = x + 1;`
`(*px)--;`	`*px = *px – 1;`	`x = x - 1;`

Наступна програма демонструє найпростіше практичне використання покажчиків, виводячи звичайну послідовність літер алфавіту:

```
#include <stdio.h>
char c;  /* змінна символьного типу*/
main()
{
  char *pc; /* покажчик на змінну символьного типу*/
  pc=&c;
  for(c='A';c<='Z';c++)
      printf("%c",*pc);
  return  0;
}
```

У операторі *printf("%c",*pc)* має місце розіменування покажчика *(*pc)* - передача у функцію значення, що зберігається за адресою, яка міститься у змінній *pc*. Щоб дійсно довести, що *pc* є псевдонімом *c*, спробуємо замінити **pc* на *c* у виклику функції – і після заміни програма працюватиме абсолютно аналогічно. Оскільки покажчики обмежені заданим типом даних, типовою серйозною помилкою їх використання буває присвоєння адреси одного типу даних покажчика іншого типу, на що компілятор реагує таким чином:

"Suspicious pointer conversion in function main()"

На *TC* це лише попередження (підозріле перетворення покажчика у функції *main()(?!)),* і якщо на нього ніяк не відреагувати, то програма працюватиме й надалі (адже помилку зафіксовано не буде) і залишається лише здогадуватися, який результат буде надалі. Зазначимо, що компілятор BC++ з приводу такого "підозрілого перетворення" пішов все-таки далі: він просто відмовляється працювати, видаючи повідомлення про помилку. Відповідальність за ініціалізацію покажчиків повністю покладається на програміста, і більш детально про це йтиметься далі.

1.8.4 Багаторівнева непряма адресація

У мові Сі можна використовувати багаторівневу непряму адресацію, тобто непряму адресацію на 1, 2 і т.д. рівні. При цьому для оголошення і звертання до значень за допомогою покажчиків можна використовувати відповідно кілька символів зірочка: *. Зірочки при оголошенні ніби уточнюють призначення імені змінної, визначаючи рівень непрямої адресації для звертання до значень за допомогою цих покажчиків. Приклад оголошення змінної і покажчиків для багаторівневої непрямої адресації можна привести наступний:

```
int      i = 123  /* де: i - ім'я змінної  */
int    *pi = &i;  /* pi - покажчик на змінну i  */
int  **ppi = &pi; /* ppi - покажчик на покажчик на
                     змінну pi */
int ***pppi = &ppi; /* pppi - покажчик на 'покажчик на
                        'покажчик на змінну ppi' */
```

Для звертання до значень за допомогою покажчиків можна прийняти наступне правило, що жорстко зв'язує форму звертання з оголошенням цих покажчиків:

- повна кількість зірочок непрямої адресації, рівна кількості зірочок при оголошенні покажчика, визначає значення змінної;
- зменшення кількості зірочок непрямої адресації додає до імені змінної слово "покажчик", причому цих слів може бути стільки, скільки може бути рівнів непрямої адресації для цих імен покажчиків, тобто стільки, скільки зірочок стоїть в оголошенні покажчика.

Наприклад, після оголошення:

```
int i, *pi=&i;
```

звертання у виді:

```
*pi - визначає значення змінної,
 pi - покажчик на змінну i.
```

А при звертанні до змінних можна використовувати різну кількість зірочок для різних рівнів адресації:

pi, ppi, pppi	– 0-й рівень адресації, пряма адресація;
*pi, *ppi, *pppi	– 1-й рівень непрямої адресації
**ppi, **pppi	– 2-й рівень непрямої адресації
***pppi	– 3-й рівень непрямої адресації

Таким чином, до покажчиків 1-го і вище рівнів непрямої адресації можливі звертання і з меншою кількістю зірочок непрямої адресації, аніж задано при оголошенні покажчика. Ці звертання визначають адреси, тобто значення покажчиків визначеного рівня адресації. Відповідність між кількістю зірочок при звертанні за допомогою покажчика і призначенням звертання за покажчиком для наведеного прикладу ілюструє таблиця 1.12 (де Р.н.а. – рівень непрямої адресації):

Таблиця 1.12. Відповідність між кількістю уточнень (*) і результатом звертання за допомогою покажчика

Звертання	*Результат звертання*	*Р.н.а.*
i	значення змінної i	1
*pi pi	значення змінної, на яку вказує pi покажчик на змінну типу int, значення pi	1 0
**ppi *ppi ppi	значення змінної типу int покажчик на змінну типу int покажчик на "покажчик на змінну типу int', значення покажчика ppi	2 1 0
***pppi **pppi *pppi pppi	значення змінної типу int; покажчик на змінну типу int покажчик на 'покажчик на змінну типу int' покажчик на 'покажчик на 'покажчик на змінну типу int', значення покажчика pppi	3 2 1 0

1.8.5 Операції над покажчиками

Мова Сі надає можливості для виконання над покажчиками операцій присвоювання, цілочисельної арифметики та порівнянь. Мовою Сі можливо:

1. присвоїти покажчику значення адреси даних, або нуль;
2. збільшити (зменшити) значення покажчика;
3. додати або відняти від значення покажчика ціле число;
4. скласти або відняти значення одного покажчика від іншого;
5. порівняти два покажчики за допомогою операцій відношення.

Змінній-покажчику можна надати певне значення за допомогою одного із способів:

1. присвоїти покажчику адресу змінної, що має місце в ОП, або нуль, наприклад:

```
pi = &j;
pi = NULL;
```

2. оголосити покажчик поза функцією (у тому числі поза *main*()) або у будь-якій функції, додавши до нього його інструкцію *static*; при цьому початковим значенням покажчика є нульова адреса (NULL);

3. присвоїти покажчику значення іншого покажчика, що до цього моменту вже має визначене значення; наприклад: pi = pj; це - подвійна вказівка однієї і тієї ж змінної;
4. присвоїти змінній-покажчику значення за допомогою функцій *calloc*() або *malloc*() - функцій динамічного виділення ОП.

Усі названі дії над покажчиками будуть наведені у прикладах програм даного розділу. Розглянемо кілька простих прикладів дій над покажчиками.

Зміну значень покажчика можна робити за допомогою операцій: +, ++, –, —. Бінарні операції (+ та –) можна виконувати над покажчиками, якщо обидва покажчики посилаються на змінні одного типу, тому що об‘єм ОП для різних типів даних може вирізнятися.

Наприклад, значення типу *int* займає 2 байти, а типу *float* - 4 байти. Додавання одиниці до покажчика додасть „квант пам'яті”, тобто кількість байтів, що займає одне значення типу, що адресується. Для покажчика на елементи масиву це означає, що здійснюється перехід до адреси наступного елемента масиву, а не до наступного байта. Тобто значення покажчика при переході від елемента до елемента масиву цілих значень буде збільшуватися на 2, а типу *float* - на 4 байти. Результат обчислення покажчиків визначений у мові Сі як значення типу *int*.

Приклад програми зміни значення покажчика на 1 квант пам'яті за допомогою операції „++”і визначення результату обчислення покажчиків даний на такому прикладі:

```
#include<stdio.h>
void main ()
{
 int a[] = { 100, 200, 300 };
 int *ptr1, *ptr2;
 ptr1=a; /*- ptrl одержує значення адреси a[0]  */
 ptr2 = &a[2]; /*- ptr2 одержує значення адреси a[2] */
 ptr1++; /* збільшення значення ptrl на квант ОП:
            ptr1 = &a[1]*/
 ptr2++; /* збільшення значення ptr2 на квант ОП:
            ptr2 = &a[3]*/
 printf (" ptr2 - ptr1 = %d\n", ptr2 - ptr1);
}
```

Результат виконання програми:

```
ptr2 – ptr1 = 2
```

Результат 2 виконання операції віднімання визначає 2 кванти ОП для значень типу int:

```
ptr2 – ptr1 = &a[3] - &a[1] = (a + 3) - (a + 1) = 2;
```

У наступному Сі-фрагменті продемонстрований приклад програми для виведення значень номерів (індексів) елементів масивів, адрес першого байта ОП для їх розміщення та значень елементів масивів. Справа в тому, що в Сі є дуже важлива властивість – ім'я масиву еквівалентно адресу його нульового елемента: x == &x[0]. Покажчики *pi* і *pf* спочатку містять значення адрес нульових елементів масивів, а при виведенні складаються з *i*-номером елемента масиву, визначаючи адресу *i*-елемента масиву. Для одержання адрес елементів масивів у програмі використовується додавання покажчиків-констант *x* та *y*, та змінних-покажчиків *pi* і *pf* з цілим значенням змінної *i*. Зміна адрес у програмі дорівнює кванту ОП для даних відповідного типу: для цілих – 2 байти, для дійсних – 4 байти.

```
#include<stdio.h>
void main()
{
  int  x[4], *pi = x, i;
  float y[4], *pf = y;
  printf("\nномер елемента адреси елементів масивів:\n"
  "i   pi+i   x + i   &x[i]   pf+i   y+i  &y[i]\n");
  for (i = 0; i < 4; i++ )
  printf("  %d :    %6u %6u %6u     %6u %6u %6u\n",
  i, pi + i, x + i, &x[i], pf + i, y + i, &y[i]);
}
```

Результати виконання програми:

номер елемента			адреси елементів масивів:			
i	pi+i.	x+i	&x[i]	pf+i	y+i	&y[i]
0:	65518	65518	65518	65498	65498	65498
1:	65520	65520	65520	65502	65502	65502
2:	65522	65522	65522	65506	65506	65506
3:	65524	65524	65524	65510	65510	65510

Мовою Сі можна визначити адреси нульового елемента масиву *x* як *x* або &x[0]: x == &x[0]. Краще і стисло використовувати просто *x* – це базова адреса масиву. Ту саму адресу елемента масиву можна представити у вигляді: x + 2 == &x[2]; x + i == &x[i].

Те саме значення можна представити у вигляді:

```
*(x + 0) == *x == x[0] -
                  значення нульового елемента масиву x;
*(x + 2) == x[2] - значення другого елемента масиву x;
*(x + i) == x[i] - значення i-го елемента масиву x.
```

А операції над елементами масиву x можна представити у вигляді:

```
*x + 2== x[0] +2;      *(x + i) - 3 == x[i] - 3;
```

1.8.6 Проблеми, пов'язані з покажчиками

Проблеми, пов'язані з покажчиками, виникають при некоректному використанні покажчиків. Усі застереження щодо некоректного використання покажчиків відносяться до мови Сі так само, як і до багатьох інших низькорівневих мов програмування. Некоректним використанням покажчиків може бути:

- спроба працювати з неініціалізованим покажчиком, тобто з покажчиком, що не містить адреси ОП, що виділена змінній;
- втрата вказівника, тобто значення покажчика через присвоювання йому нового значення до звільнення ОП, яку він адресує;
- незвільнення ОП, що виділена за допомогою функції *malloc();*
- спроба повернути як результат роботи функції адресу локальної змінної класу auto (про функції та класи змінних йтиметься далі);

Запит на виділення ОП з купи робиться за допомогою функцій *calloc()* та *malloc()*. Повернення (звільнення) ОП робиться за допомогою функції *free()*. Розглянемо деякі проблеми, пов'язані з покажчиками.

При оголошенні покажчика на скалярне значення будь-якого типу оперативна пам'ять для значення, що адресується, не резервується. Виділяється тільки ОП для змінної-покажчика, але покажчик при цьому не має значення. Якщо покажчик має специфікатор static, то ініціюється початкове значення покажчика, рівне нулю (особливості статичних змінних, про що йтиметься в окремому розділі). Приклад ініціалізації покажчиків нульовими значеннями при їх оголошенні:

```
static int *pi, *pj; /* pi = NULL; pj= NULL; */
```

Розглянемо приклад, що містить грубу помилку: спробу працювати з непроініціалізованим покажчиком.

```
int *x; /* змінній-покажчику 'x' виділена ОП, але  'x'
           не містить значення адреси ОП для змінної */
*x = 123; /* - груба помилка! */
```

Таке присвоювання помилкове, тому що змінна-покажчик x не має значення адреси, за яким має бути розташоване значення змінної.

Компілятор видасть попередження:

Warning: Possible use of 'x' before definition

При цьому випадкове (непроініціалізоване) значення покажчика (сміття) може бути неприпустимим адресним значенням! Наприклад, воно може збігатися з адресами розміщення програми або даних користувача, або даних операційної системи. Запис цілого числа 123 за такою адресою може порушити працездатність програми користувача або самої ОС. Компілятор не виявляє цю помилку, це повинен робити програміст!

Виправити ситуацію можна за допомогою функції *malloc()*. Форма звертання до функції *malloc()* наступна:

ім‘я-покажчика = (тип-покажчика) malloc (об’єм -ОП) ;

де ім‘я-покажчика - ім'я змінної-покажчика, тип-покажчика - тип значення, що повертається функцією malloc;

об‘єм-ОП - кількість байтів ОП, що виділяються змінній, яка адресується.

Наприклад:

```
x = (int *) malloc ( sizeof (int) );
```

При цьому з купи виділяється 2 байти ОП для цілого значення, а отримана адреса його розміщення заноситься в змінну-покажчик *x*. Значення покажчика гарантовано не збігається з адресами, що використовуються іншими програмами, у тому числі програмами ОС. Параметр функції *malloc* визначає об‘єм ОП для цілого значення за допомогою функції *sizeof(int)*. Запис (int *) означає, що адреса, що повертається функцією *malloc()*, буде розглядатися як покажчик на змінну цілого типу. Це операція приведення типів.

Таким чином, помилки не буде у випадку використання наступних операторів:

```
int *x;    /* x - ім'я покажчика, він одержав ОП  */
x = (int *) malloc ( sizeof(int));
/* Виділена ОП цілому значенню, на яке вказує `x` */
*x = 123; /* змінна, на яку вказує 'x', одержала
             значення 123*/
```

Повернення (звільнення) ОП у купі виконує функція free(). Її аргументом є ім'я покажчика, що посилається на пам'ять, що звільняється. Наприклад:

```
free (x);
```

Щоб уникнути помилок при роботі з функціями не слід повертати як результат їхнього виконання адреси автоматичних (локальних) змінних функції. Оскільки при виході з функції пам'ять для всіх автоматичних змінних звільняється, повернута адреса може бути використаною системою й інформація за цією адресою може бути невірною. Можна повернути адресу ОП, що виділена з купи.

Одна з можливих помилок - подвійна вказівка на дані, розташовані у купі, і зменшення об‘єму доступної ОП через незвільнення отриманої ОП. Це може бути для будь-якого типу даних, у тому числі для скаляра або масиву. Розглянемо випадок для скаляра.

Приклад фрагмента програми з подвійною вказівкою і зменшенням об‘єму доступної ОП через незвільнення ОП наведений нижче:

```
#include<alloc.h>
void main ()
{
  /* Виділення ОП динамічним змінним x, y и z: */
  int *x = (int *) malloc ( sizeof(int)),
      *y = (int *) malloc ( sizeof(int)),
      *z = (int *) malloc ( sizeof(int));
  /* Ініціалізація значення покажчиків x, y, z;*/
  *x = 14; *y = 15; *z = 17;
  /*Динамічні змінні одержали конкретні цілі значення*/
  y=x; /* груба помилка - втрата покажчика на динамічну
          змінну в без  попереднього звільнення її ОП */
}
```

У наведеному вище прикладі немає оголошення імен змінних, є тільки покажчики на ці змінні. Після виконання оператора y = x; *x* та *y* є двома покажчиками на ту саму ОП змінної *x. Тобто *x = 14; і *y = 14. Крім того, 2 байти, виділені змінній, яку адресував *y* для розміщення цілого значення (*y), стають недоступними (загублені), тому що значення y, його адреса, замінені значенням x. А в купі ці 2

байти для *y вважаються зайнятими, тобто розмір купи зменшений на 2 байти. Відбулося зменшення доступної ОП. Цього слід уникати.

Щоб уникнути такої помилки треба попередньо звільнити ОП, виділену змінній *y, а потім виконати присвоювання значення змінній y. Наприклад:

```
free (y);  /* звільнення ОП, виділеної змінної '*y' */
y = x;   /* присвоювання нового значення змінній 'y' */
```

Чи можна змінній-покажчику присвоїти значення адреси в операторі оголошення ? Наприклад:

```
int *x = 12345;
```

Тут константа 12345 цілого типу, а значенням покажчика x може бути тільки адресою, покажчиком на байт в ОП. Тому компілятор при цьому видасть повідомлення про помилку:

Error PR.CPP 3: Cannot convert 'int to 'int *'

Проте не викличе помилки наступне присвоювання:

```
int a[5], *x = a;
```

Використання покажчиків часто пов'язано з використанням масивів різних типів. Кожний з типів даних масивів має свої особливості. Тому далі розглянемо властивості покажчиків для роботи з масивами.

1.9 Масиви

1.9.1 Основні поняття

Між покажчиками і масивами існує тісний взаємозв'язок. Будь-яка дія над елементами масивів, що досягається індексуванням, може бути виконана за допомогою покажчиків (посилань) і операцій над ними. Варіант програми з покажчиками буде виконаний швидше, але для розуміння він складніший.

Як показує практика роботи на Сі, покажчики рідко використовуються зі скалярними змінними, а частіше – з масивами. Покажчики дають можливість застосовувати адреси приблизно так, як це робить ЕОМ на машинному рівні. Це дозволяє ефективно організувати роботу з масивами. Будь-яку серйозну програму, що використовує масиви, можна написати за допомогою покажчиків.

Для роботи з масивом необхідно:

1. визначити ім'я масиву, його розмірність (кількість вимірів) і розмір – кількість елементів масиву;

2. виділити ОП для його розміщення.

У мові Сі можна використовувати масиви даних будь-якого типу:

- статичні: з виділенням ОП до початку виконання функції; ОП виділяється в стеку або в ОП для статичних даних;
- динамічні: ОП виділяється з купи в процесі виконання програми, за допомогою функцій *malloc*() і *calloc*().

Динамічні змінні використовують, якщо розмір масиву невідомий до початку роботи програми і визначається в процесі її виконання, наприклад за допомогою обчислення або введення.

Розмір масиву визначається:

1. для статичних масивів при його оголошенні; ОП виділяється до початку виконання програми; ім'я масиву - покажчик-константа; кількість елементів масиву визначається:
   a. явно; наприклад: int a[5];
   b. неявно, при ініціалізації елементів масиву; наприклад:
   ```
   int a[] = { 1, 2, 3 };
   ```
2. для динамічних масивів у процесі виконання програми; ОП для них запитується і виділяється динамічно, з купи; ім'я покажчика на масив - це змінна; масиви ці можуть бути:
   a. одновимірні і багатовимірні; при цьому визначається кількість елементів усього масиву й ОП запитується для всього масиву;
   b. вільні (спеціальні двовимірні); при цьому визначається кількість рядків і кількість елементів кожного рядка, і ОП запитується і виділяється для елементів кожного рядка масиву в процесі виконання програми; при використанні вільних масивів використовують масиви покажчиків;

Розмір масиву можна не вказувати. В цьому разі необхідно вказати порожні квадратні дужки:

1. якщо при оголошенні ініціалізується значення його елементів; наприклад:
   ```
   static int a[] = {1, 2, 3};
   char b[] = "Відповідь:";
   ```
2. для масивів - формальних параметрів функцій; наприклад:
   ```
   int fun1(int a[], int n);
   int fun2(int b[k][m][n]);
   ```

3. при посиланні на раніше оголошений зовнішній масив; наприклад:

```
int a[5];  /* оголошення зовнішнього масиву  */
main ()
{
 extern int a[];/*посилання на зовнішній масив */
}
```

В усіх оголошеннях масиву ім'я масиву - це покажчик-константа! Для формування динамічного масиву може використовуватися тільки ім'я покажчика на масив – це покажчик-змінна. Наприклад:

```
int   *m1 = (int * ) malloc ( 100 * sizeof (int)) ;
float *m2 = (float * ) malloc ( 200 * sizeof (float)) ;
```

де *m1* - змінна-покажчик на масив 100 значень типу *int*;

m2 - змінна-покажчик на масив 200 значень типу *float*.

Звільнення виділеної ОП відбувається за допомогою функції:

```
free (покажчик-змінна) ;
```

Наприклад:

```
free(ml);
free(m2);
```

Звертання до елементів масивів *m1* і *m2* може виглядати так:

```
m1[i], m2[j].
```

Пересилання масивів у Сі немає. Але можна переслати масиви поелементно або сумістити масиви в ОП, давши їм практично те саме ім'я.

Наприклад:

```
int *m1 = (int *) malloc(100 * sizeof(int));
int *m2 = (int *) malloc(100 * sizeof(int));
```

Для пересилання елементів одного масиву в інший можна використати оператор циклу:

```
for (i = 0; i < 100; i++ ) m2[i] = ml [i] ;
```

Замість m2[i] = m1 [i]; можна використовувати:

*m2++ = *ml++; або: *(m2 + i) = *(ml + i) ;

За допомогою покажчиків можна сполучити обидва масиви й у такий спосіб:

```
free(m2);
m2 = ml ;
```

Після цього обидва масиви займатимуть одну й ту саму область ОП, виділену для масиву m1. Однак це не завжди припустимо. Наприклад, коли масиви розташовані в різних типах ОП: один - у стеку, інший – у купі. Наприклад, у функції *main()* оголошені:

```
int *m1 = (int *) malloc(100* sizeof(int));
int m2[100] ;
```

У вищенаведеному прикладі m1 – пакажчик-змінна, і масив *m1* розташований у купі, m2 - покажчик-константа, і масив *m2* розташований у стеку. У цьому випадку помилковий оператор: *m2 = m1*; тому що *m2* - це покажчик-константа. Але після *free(m1)* припустимим є оператор:

```
m1 = m2; /* оскільки m1 - покажчик-змінна */
```

Для доступу до частин масивів і до елементів масивів використовується індексування (індекс). Індекс - це вираз, що визначає адресу значення або групи значень масиву, наприклад адреса значень чергового рядка двовимірного масиву. Індексування можна застосовувати до покажчиків-змінних на одновимірний масив - так само, як і до покажчиків-констант.

Індексний вираз обчислюється шляхом додавання адреси початку масиву з цілим значенням для одержання адреси необхідного елемента або частини масиву. Для одержання значення за індексним виразом до результату – адреси елемента масиву застосовується операція непрямої адресації (*), тобто одержання значення за заданою адресою. Відповідно до правил обчислення адреси цілочисельний вираз, що додається до адреси початку масиву, збільшується на розмір кванта ОП типу, що адресується покажчиком.

Розглянемо способи оголошення і формування адрес частини масиву й елементів одновимірних і багатомірних масивів за допомогою покажчиків.

1.9.2 Оголошення та звертання в одновимірних масивах

Форма оголошення одновимірного масиву з явною вказівкою кількості елементів масиву:

тип ім‘я_масива [кількість-елементів-масива];

Звертання до елементів одновимірного масиву в загальному випадку можна представити індексуванням, тобто у вигляді

ім‘я-масива [вираз];

де ім‘я-масиву - покажчик-константа;

вираз – індекс, число цілого типу; він визначає зсув - збільшення адреси заданого елемента масиву щодо адреси нульового елемента масиву.

Елементи одновимірного масиву розташовуються в ОП підряд: нульовий, перший і т д. Приклад оголошення масиву:

```
int a[10];
int *p = a;          /* - p одержує значення  a */
```

При цьому компілятор виділяє масив в стеку ОП розміром *(sizeof(Type) * розмір-масиву)* байтів.

У вищенаведеному прикладі це 2 * 10 = 20 байтів. Причому *a* - покажчик-константа, адреса початку масиву, тобто його нульового елемента, *p* - змінна; змінній *p* можна присвоїти значення одним із способів:

```
p = a;
p = &a[0];
p = &a[i];
```

де &a[i] == (a + i) - адреса *i*-елемента масиву.

Відповідно до правил перетворення типів значення адреси *i*-елемента масиву на машинному рівні формується таким чином:

```
&a[i]= a + i * sizeof(int);
```

Справедливі також наступні співвідношення:

`&a == a+0 == &a[0]` – адреса a[0] - нульового елемента масиву;
`a+2 == &a[2]` – адреса a[2] - другого елементи масиву;
`a+i == &a[i]` – адреса a[i] - i-го елемента масиву;
`*a==*(a+0)==*(&a[0])==a[0]` – значення 0-ого елемента масиву;
`*(a + 2) == a[2]` – значення a[2] - другого елементи масиву;
`*(a + i) == a[i]` – значення a[i] - i-го елемента масиву;
`*a + 2 == a[0] + 2` – сума значень a[0] і 2.

Якщо *p* - покажчик на елементи такого ж типу, які і елементи масиву *a* та *p=a*, то *a* та *p* взаємозамінні; при цьому:

```
p == &a[0] == a + 0;
p+2 == &a[2] == a + 2;
*(p + 2) == (&a[2]) == a[2] == p[2];
*(p + i) == (&a[i]) == a[i] == p[i];
```

Для *a* та *p* еквівалентні всі звертання до елементів *a* у вигляді:

```
a[i], *(a+i), *(i+a), i[a], та
p[i], *(p+i), *(i+p), i[p]
```

1.9.3 Оголошення та звертання до багатовимірних масивів

У даному розділі розглянемо оголошення і зв'язок покажчиків і елементів багатомірних масивів - що мають 2 та більше вимірів.

Багатомірний масив у мові Сі розглядається як сукупність масивів меншої розмірності. Наприклад, двовимірний масив - це сукупність одновимірних масивів (його рядків), тривимірний масив - це сукупність матриць, матриці - сукупності рядків, а рядок - сукупність елементів одновимірного масиву.

Елементи масивів розташовуються в ОП таким чином, що швидше змінюються самі праві індекси, тобто елементи одновимірного масиву розташовуються підряд, двовимірного - по рядках, тривимірного - по матрицях, а матриці - по рядках.

Для звертання до елементів багатомірного масиву можна використовувати нуль і більш індексів (індексних виразів):

ім‘я-масиву [вираз1][вираз2] ...

Наприклад, для звертання:

- до одновимірного масиву можна використовувати одно-індексний вираз (індекс);
- до двовимірного – 1 або 2 індексний вираз;
- до тривимірного – 1, 2 або 3 індексний вираз і т.д.

При звертанні до багатомірних масивів одержання значення елемента масиву можливо тільки після визначення адреси елемента масиву, тобто при повній кількості індексів. При цьому обчислюються індексні вираз зліва на право, і доступу до значення виконується після обчислення останнього індексного виразу.

Приклад оголошення двовимірного масиву значень типу *int*:

int a[m][n] ;

Цей масив складається з *m* одновимірних масивів (рядків), у кожному з яких утримується *n* елементів (стовпців). При роботі з цим двовимірним масивом можна використовувати одно або 2 індексний вираз. Наприклад:

a[i][j]- містить 2 індекси; використовується для звертання до елемента і-рядка, j-стовпця масиву; обчислюються індексні вирази, визначається адреса елемента масиву і вилучається його значення;

a[i] - містить 1 індекс; визначає адресу одновимірного масиву: адреса початку і-рядка масиву;

a - не містить індексу і визначає адресу масиву, його нульового елемента.

Таким чином, звертання до двовимірних масивів за допомогою імені і тільки одного індексу визначає покажчик на початок відповідного рядка масиву (адреса його нульового елемента). Наприклад:

```
a[0] == &a[0][0] == a+0*n*sizeof(int);
a[1] == &a[1][0] == a+1*n*sizeof(int);
a[i] == &a[i][0] == a+i*n*sizeof(int);
```

Приклад оголошення тривимірного масиву:

```
int a[k][m][n] ;
```

де:

- *k*- кількість матриць з m рядками і n стовпцями;

- *m* - кількість рядків (одновимірних масивів) у матриці;

- *n* - кількість стовпців (елементів у рядку) матриці.

Цей масив складається з *k* матриць, кожна з яких складається з m одновимірних масивів (рядків) по n елементів (стовпців). При звертанні до цього масиву можна використовувати імена:

a[l][i][j] - містить 3 індекси; використовується для звертання до елемента l-матриці, i-рядка. j-стовпця масиву; обчислюються індексні вирази, визначається адреса елемента масиву і вилучається його значення;

a[k][i] - визначає одновимірний масив - адреса початку i-рядка; k - матриці;

a[k] - визначає двовимірний масив - адреса початку k - матриці, тобто нульового елемента його нульового рядка;

a - адреса початку масиву, нульового елемента нульового рядка нульової матриці.

Наприклад:

```
int b[3][4][5];
int i, *ip, *ipp;
i = b[0][0][1];
ip = b[2][0];
ipp = b[2];
```

де: *ip*, *ipp* - покажчики на значення типу *int*.

Після ip = b[2][0]; *ip* є покажчиком на елемент 0-рядка 0-го стовпця 2-й матриці масиву, тобто b[2][0][0].

Після *ipp* = b[2]; ipp адресує 0-й рядок 2-ї матриці масиву, тобто містить адреса b[2][0][0].

Звертання до елементів багатомірного масиву більш детально розглянемо на прикладі двовимірного масиву. Наприклад:

```
int a[3][4];        /* a - покажчик-константа */
int *p = a;         /* p – покажчик-змінна */
```

Після цього покажчик p можна використовувати замість покажчика *a* для звертання до рядків або елементів масиву *a* у вигляді: ім'я покажчика і зсув елемента щодо адреси початку масиву *a*.

В ОП елементи масиву *a* розташовуються таким чином, що швидше всіх змінюється самий правий індекс, тобто в послідовності:

a[0][0] a[0][1] a[0][2] a[0][3] a[1][0] ... a[2][2] a[2][3].

При цьому для звертання до масиву *a* можна використовувати імена:

&a == a == &a[0][0] == *a

адреса a[0][0] - елемента 0-ого рядка 0-ого стовпця масиву a;

****a == *(&a[0][0]) == a[0][0]**

значення елемента нульового рядка нульового стовпця масиву a;

a[i] == (a + i) == *(a + i) == &a[i][0]

адреса елемента i-рядка 0-стовпця;

***a[i] == **(a + i) == *(&a[i]) == a[i][0]**

значення 0-го елемента i-рядка;

a[i][j] == *(*(a + i) + j) == *(a[i] + j) == a[i][j]

значення елемента i-рядка j-стовпця масиву a;

де:

(a + i) == *(a + i) == a[i]

адреса 0-го елемента i-рядка == &a[i][0];

(*(a + i) + j)

адреса j-елемента i-рядка = &a[i][j];

***(*(a + i) + j)**

значення j-елемента i-рядка = a[i][j].

Значення адреси початку i-рядка (адреси 0-елемента i-рядка) на машинному рівні формується у виді:

a[i] = a + i == (a+i*n*sizeof(int)), де n - кількість значень в одному рядку.

Таким чином, адреса (i+1)-рядка відстоїть від i-рядка на (n*sizeof(int)) байтів, тобто на відстань одного рядка масиву.

Вираз a[i][j] компілятор Сі переводить в еквівалентний вираз:

*(*a + i) + j). Зрозуміло, запис a[i][j] більш традиційний у математиці і більш наочний.

До елементів двовимірного масиву можна звернутися і за допомогою скалярного покажчика на масив. Наприклад, після оголошення:

```
int a[m][n], *p = a;
```

`*(p+i*n+j)` - значення j - елемента і-рядка ;

де: n - кількість елементів у рядку;

i*n + j - змішання a[i][j]- елемента відносно початку масиву a.

1.10 Масиви покажчиків

За допомогою масивів покажчиків можна формувати великі масиви і вільні масиви - колекції масивів будь-яких типів.

1.10.1 Робота з великими масивами

Розмір одного масиву даних повинний бути не більше 64 Кб. Але в реальних задачах можуть використовуватися масиви, що вимагають ОП, більшої ніж 64 Кб. Наприклад, масив даних типу *float* з 300 рядків і 200 стовпців потребує для розміщення 300 * 200 * 4 = 240000 байтів.

Для вирішення поставленої задачі можна використовувати масив покажчиків і динамічне виділення ОП для кожного рядка матриці. Рядок матриці не повинен перевищувати 64 Кб. У вищенаведеному прикладі ОП для рядка складає всього 800 байтів. Для виділення ОП з купи кожен рядок повинний мати покажчик. Для всіх рядків масиву треба оголосити масив покажчиків, по одному для кожного рядка. Потім кожному рядку масиву виділити ОП, привласнивши кожному елементу масиву покажчиків адресу початку розміщення рядка в ОП, і заповнити цей масив.

У запропонованому лістингу представлена програма для роботи з великим масивом цілих значень: з 300 рядків і 200 стовпців. Для розміщення він вимагає: 200 * 300 * 2 = 120000 байтів. При формуванні великого масиву використовується p - статичний масив покажчиків

При виконанні програми перебираються *i*-номери рядків масиву. Для кожного рядка за допомогою функції *malloc()* виконується запит ОП з купи і формується p[i] - значення покажчика на дані і-рядки. Потім перебираються і-номери рядків від 1 до 200. Для кожного рядка перебираються j-номери стовпчиків від 1 до 300. Для кожного i та j за допомогою генератора випадкових чисел формуються і виводяться *(p[i] + j) - значення елементів масиву. Після обробки масиву за допомогою функції free(p[i]) звільняється ОП виділена і-рядку масиву.

У наведеній нижче програмі використовуються звертання до $A_{i,j}$ - елементів масиву у вигляді: *(p[i]+j), де p[i] +j – адреса $A_{i,j}$-елемента масиву.

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
void main()
{
  int *p[200], i, j;
  clrscr();
  randomize();
  for (i=0;i<200;i++)
  /* Запит ОП для рядків великого масиву: */
  p[i] = (int*) malloc (300 * sizeof (int));
  for (i = 0; i < 200; i++)
  for (j = 0; j < 300; j++ )
  {
    *(p[i] + j ) = random(100);
    printf("%3d", *(p[i] + j ));
    if ( (j + 1) % 20 == 0 )
       printf ("\n" ) ;
  }
  /* Звільнення ОП рядків великого масиву: */
  for ( i=0; i < 200; i++ )
    free( p[i] );
}
```

У програмі використовується *p* - масив покажчиків.

1.10.2 Вільні масиви та покажчики

Термін „вільний” масив відносять до двовимірних масивів. Вони можуть бути будь-якого типу, у тому числі *int*, *float*, *char* і типу структура. Вільний масив - це двовимірний масив, у якому довжини його рядків можуть бути різними. Для роботи з вільними масивами використовуються масиви покажчиків, що містять в собі кількість елементів, рівну кількості рядків вільного масиву. Кожен елемент масиву покажчиків містить адресу початку рядка значень вільного масиву. ОП виділяється для кожного рядка вільного масиву, наприклад за допомогою функції *malloc(),* і звільняється функцією *free()*. Для того щоб виконати функцію *malloc(),* треба визначити кількість елементів у рядку, наприклад із вводу користувача або яким-небудь іншим способом. У нульовому елементі кожного рядка вільного масиву зберігається число, рівне кількості елементів даного рядка Дані в

кожен рядок можуть вводитися з файлу або з клавіатури в режимі діалогу. Приклад вільного масиву цілих чисел приведений на рис 1.12:

j i	Кількість 0	1	2	3	4
0	1	3			
1	4	1	2	3	4
2	2	5	6		

Рис. 1.12. Схема представлення вільного масиву цілих значень

У масиві на рис. 1.12 три рядки; у нульовому стовпці кожного рядка стоїть кількість елементів даного рядка. Далі - значення елементів матриці.

Приклад оголошення вільного масиву цілих, тобто статичного масиву покажчиків на дані типу *int:*

```
int *a[100];
```

Для масиву a приділяється ОП для 100 покажчиків на значення цілого типу, по одному покажчику на кожний з 100 рядків вільного масиву. Після визначення кількості елементів рядка для значень рядка повинна бути виділена ОП і сформоване значення покажчика в змінній a[i]. Цей покажчик посилається на область ОП, виділену для значень і-рядка матриці. Тільки після цього можна заносити в цю ОП значення елементів вільного масиву.

Реально ОП - це лінійна послідовність перенумерованих байтів. Елементи рядків вільного масиву можуть бути розташовані підряд або несуміжними відрізками ОП, виділеними для рядків.

1.11 Символьні рядки

1.11.1 Основні відомості про представлення рядків

Символьний рядок представляє собою набір з одного або більше символів.

Приклад *:* **`"Це рядок".`**

В мові Сі немає спеціального типу даних, який можна було б використовувати для опису рядків. Замість цього рядки представляються у вигляді масиву елементів типу *char*. Це означає, що символи рядка розташовуються в пам'яті в сусідніх комірках, по одному символу в комірці.

Ц	е		р	я	д	о	к	\0

Рис. 1.13. Представлення рядка у вигляді масиву символів

Необхідно відмітити, що останнім елементом масиву є символ ‘\0’. Це нульовий символ (байт, кожний біт якого рівний нулю). У мові Сі він використовується для того, щоб визначати кінець рядка.

Примітка. Нульовий символ – це не цифра 0; він не виводиться на друк і в таблиці символів ASCII (див. додаток) має номер 0. Наявність нульового символу передбачає, що кількість комірок масиву повинна бути принаймні на одну більше, ніж число символів, які необхідно розміщувати в пам’яті. Наприклад, оголошення

```
char str[10];
```

передбачає, що рядок містить може містити максимум 9 символів.

Основні методи ініціалізації символьних рядків.

- ```
  char str1[]= "ABCdef";
  ```
- ```
  char str2[]={'A', 'B', 'C', 'd', 'e', 'f',0};
  ```
- ```
  char str3[100];
  gets(str3);
  ```
- ```
  char str4[100];
  scanf("%s",str4);
  ```

Усі константи-рядки в тексті програми, навіть ідентично записані, розміщуються за різними адресами в статичній пам’яті. З кожним рядком пов’язується сталий покажчик на його перший символ. Власне, рядок-константа є виразом типу „покажчик на *char*” зі сталим значенням – адресою першого символу.

Так, присвоювання p=“ABC” (p – покажчик на *char*) встановлює покажчик *p* на символ ‘A’; значенням виразу *(“ABC”+1) є символ ‘B’.

Елементи рядків доступні через покажчики на них, тому будь-який вираз типу „покажчик на *char*” можна вважати рядком.

Необхідно мати також на увазі те, що рядок вигляду „x” – не те ж саме, що символ ‘x’. Перша відмінність : ‘x’ – об’єкт одного з основних типів даних мови Сі (*char*), в той час, як „x” – об’єкт похідного типу (масиву елементів типу *char*). Друга різниця : „x” насправді складається з двох символів – символу ‘x’ і нуль-символу.

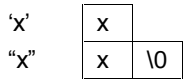


Рис. 1.14. Різниця між представленням ‘x’ та “x”

1.11.2 Функції роботи з рядками

1. Функції введення рядків.

Прочитати рядок із стандартного потоку введення можна за допомогою функції *gets*(). Вона отримує рядок із стандартного потоку введення. Функція читає символи до тих пір, поки їй не зустрінеться символ нового рядка ‘\n’, який генерується натисканням клавіші ENTER. Функція зчитує всі символи до символу нового рядка, додаючи до них нульовий символ ‘\0’.

Синтаксис :

```
char *gets(char *buffer);
```

Як відомо, для читання рядків із стандартного потоку введення можна використовувати також функцію *scanf*() з форматом %s. Основна відмінність між *scanf*() і *gets*() полягає у способі визначенні досягнення кінця рядка; функція *scanf*() призначена скоріше для читання слова, а не рядка. Функція *scanf*() має два варіанти використання. Для кожного з них рядок починається з першого не порожнього символу. Якщо використовувати %s, то рядок продовжується до (але не включаючи) наступного порожнього символу (пробіл, табуляція або новий рядок). Якщо визначити розмір поля як %10s, то функція *scanf*() не прочитає більше 10 символів або ж прочитає послідовність символів до будь-якого першого порожнього символу.

2. Функції виведення рядків.

Тепер розглянемо функції *виведення рядків*. Для виведення рядків можна використовувати функції *puts*() і *printf*().

Синтаксис функції *puts*():

```
int puts(char *string);
```

Ця функція виводить всі символи рядка *string* у стандартний потік виведення. Виведення завершується переходом на наступний рядок.

Різниця між функціями *puts*() і *printf*() полягає в тому, що функція *printf*() не виводить автоматично кожний рядок з нового рядка.

Стандартна бібліотека мови програмування Сі містить клас функцій для роботи з рядками, і всі вони починаються з літер *str*. Для того, щоб використовувати одну або декілька функції необхідно підключити файл *string.h*.

```
#include<string.h>
```

3. Визначення довжини рядка. Для визначення довжини рядка використовується функція *strlen*(). Її синтаксис :

size_t strlen(const char *s);

Функція strlen() повертає довжину рядка s, при цьому завершуючий нульовий символ не враховується.

Приклад **:**

```
char *s= "Some string";
int len;
```

Наступний оператор встановить змінну len рівною довжині рядка, що адресується покажчиком **s**:

```
len = strlen(s); /* len == 11 */
```

4. Копіювання рядків. Оператор присвоювання для рядків не визначений. Тому, якщо *s1* і *s2* – символьні масиви, то неможливо скопіювати один рядок в інший наступним чином.

```
char s1[100];
char s2[100];
s1 = s2; /* помилка */
```

Останній оператор (*s1=s2;*) не скомпілюється.

Щоб скопіювати один рядок в інший необхідно викликати функцію копіювання рядків *strcpy*(). Для двох покажчиків *s1* і *s2* типу char * оператор

```
strcpy(s1,s2);
```

копіює символи, що адресуються покажчиком *s2* в пам'ять, що адресується покажчиком *s1*, включаючи завершуючі нулі.

Для копіювання рядків можна використовувати і функцію *strncpy*(), яка дозволяє обмежувати кількість символів, що копіюються.

```
strncpy(destantion, source, 10);
```

Наведений оператор скопіює 10 символів із рядка *source* в рядок *destantion*. Якщо символів в рядку *source* менше, ніж вказане число

символів, що копіюються, то байти, що не використовуються встановлюються рівними нулю.

Примітка. Функції роботи з рядками, в імені яких міститься додаткова літера *n* мають додатковий числовий параметр, що певним чином обмежує кількість символів, з якими працюватиме функція.

5. Конкатенація рядків. Конкатенація двох рядків означає їх об'єднання, при цьому створюється новий, більш довгий рядок. Наприклад, при оголошенні рядка

```
char first[]= "Один ";
```

оператор

```
strcat(first, „два три чотири!");
```

перетворить рядок *first* в рядок "Один два три чотири".

При викликанні функції *strcat*(s1,s2) потрібно впевнитися, що перший аргумент типу char * ініціалізований і має достатньо місця щоб зберегти результат. Якщо *s1* адресує рядок, який вже записаний, а *s2* адресує нульовий рядок, то оператор

```
strcat(s1,s2);
```

перезапише рядок *s1*, викликавши при цьому серйозну помилку.

Функція *strcat*() повертає адресу рядка результату (що співпадає з її першим параметром), що дає можливість використати „каскад" декількох викликів функцій :

```
strcat(strcat(s1,s2),s3);
```

Цей оператор додає рядок, що адресує *s2*, і рядок, що адресує *s3*, до кінця рядка, що адресує *s1*, що еквівалентно двом операторам:

```
strcat(s1,s2);
strcat(s1,s3);
```

Повний список прототипів функцій роботи з рядками можна знайти в додатках на стор.

6. Порівняння рядків. Функція *strcmp*() призначена для порівняння двох рядків. Синтаксис функції :

```
int strcmp(const char *s1, const char*s2);
```

Функція *strcmp*() порівнює рядки *s1* і *s2* і повертає значення 0, якщо рядки рівні, тобто містять одне й те ж число однакових символів. При порівнянні рядків ми розуміємо їх порівняння в лексикографічному порядку, приблизно так, як наприклад, в словнику. У функції насправді здійснюється посимвольне порівняння рядків.

Кожний символ рядка *s1* порівнюється з відповідним символом рядка *s2*. Якщо *s1* лексикографічно більше *s2*, то функція *strcmp*() повертає додатне значення, якщо менше, то – від'ємне.

1.12 Основні методи сортування масивів

Починаючи з даного розділу, розглянемо декілька методів впорядкування елементів масиву, які широко використовуються у практичному програмуванні.

1.12.1 Метод бульбашкового сортування

Метод „бульбашкового сортування" ґрунтується на перестановці сусідніх елементів. Для впорядкування елементів масиву здійснюються повторні проходи по масиву.

Переміщення елементів масиву здійснюється таким чином : масив переглядається зліва направо, здійснюється порівняння пари сусідніх елементів; якщо елементи в парі розміщені в порядку зростання, вони лишаються без змін, а якщо ні – міняються місцями.

В результаті першого проходу найбільше число буде поставлено в кінець масиву. У другому проході такі операції виконуються над елементами з першого до (N-1)-ого, у третьому – від першого до (N-2)-ого і т.д. Впорядкування масиву буде закінчено, якщо при проході масиву не виконається жодної перестановки елементів масиву. Факт перестановки фіксується за допомогою деякої змінної (у наступному прикладі – *is)*, яка на початку має значення 0 і набуває значення 1 тоді, коли виконається перестановка в якій-небудь парі.

Масив до впорядкування	22	20	–1	–40	88	–75	–22
Перший перегляд масиву	20	–1	–40	22	–75	–22	88
Другий перегляд масиву	–1	–40	20	–75	–22	22	88
Третій перегляд масиву	–40	–1	–75	–22	20	22	88
Четвертий перегляд масиву	–40	-75	–22	–1	20	22	88
П'ятий перегляд масиву	–75	-40	–22	–1	20	22	88

Рис. 1.15. Бульбашкове сортування

```
const n=10;
int a[n], i, c, is;
/* … */
do {
  is=0;
  for (i=1;i<n;i++)
   if (a[i-1]>a[i])
   {
     c=a[i];
     a[i]=a[i-1];
     a[i-1]=c;
     is=1;
   };
} while (is);
```

1.12.2 Сортування методом вибору

Даний метод сортування передбачає наступні дії : масив переглядається перший раз, знаходиться мінімальний елемент цього масиву, який міняється місцями з першим елементом. Другий раз масив переглядається, починаючи з другого елементу. Знову знаходиться мінімальний елемент, який міняється місцями з другим елементом масиву.

Даний процес виконується до тих пір, поки не буде поставлено на місце N–1 елемент.

Масив до впорядкування	22	20	–1	–40	88	–75	–22
Перший перегляд масиву	**–75**	20	–1	–40	88	22	–22
Другий перегляд масиву	**–75**	**–40**	–1	20	88	22	–22
Третій перегляд масиву	**–75**	**–40**	**–22**	20	88	22	–1
Четвертий перегляд масиву	**–75**	**–40**	**–22**	**–1**	88	22	20
П’ятий перегляд масиву	**–75**	**–40**	**–22**	**–1**	**20**	22	88
Шостий перегляд масиву	**–75**	**–40**	**–22**	**–1**	**20**	**22**	88

Рис. 1.16. Сортування методом вибору

```
const int n=20;
int b[n];
int imin, i, j, a;
/* … */
for (i=0;i<n-1;i++)
{
  imin=i;
  for (j=i+1;j<n;j++)
   if (b[j]<b[imin]) imin=j;
  a=b[i];
  b[i]=b[imin];
  b[imin]=a;
}
```

1.12.3 Сортування вставками

Даний метод сортування називається сортування вставками, так як на *i*-му етапі відбувається „вставка” *i*-ого елемента a[i] в потрібну позицію серед елементів a[1], a[2], …, a[i-1], які вже впорядковані. Після цієї вставки перші *i* елементів будуть впорядковані.

Саме таким способом звичайно сортують карти, тримаючи в лівій руці вже впорядковані карти, і взявши правою рукою чергову карту вставляють її в потрібне місце, порівнюючи її з іншими проходячи справа наліво.

Масив до впорядкування	**22**	20	–1	–40	88	–75	–22
Перший перегляд масиву	**20**	**22**	-1	-40	88	-75	-22
Другий перегляд масиву	**-1**	**20**	**22**	-40	88	-75	-22
Третій перегляд масиву	**-40**	**-1**	**20**	**22**	88	-75	-22
Четвертий перегляд масиву	**-40**	**-1**	**20**	**22**	**88**	-75	-22
П’ятий перегляд масиву	**-75**	**-40**	**-1**	**20**	**22**	**88**	-22
Шостий перегляд масиву	**-75**	**-40**	**-22**	**-1**	**20**	**22**	**88**

Рис. 1.17. Сортування вставками

Реалізувати сортування масиву вставками можна так :

```
const int n=20;
int b[n];
int i,j,c;
/* … */
for (i=1;i<n;i++)
{
  c=a[i];
  for (j=i-1;j>=0&&a[j]>c;j--)
  a[j+1]=a[j];
  a[j+1]=c;
}
```

1.12.4 Швидке сортування

Швидке сортування полягає в тому, що множина елементів B { k_1, k_2, …, k_n } перетворюється на множину B_1, {k_1}, B_2, де B_1 – підмножина B з елементами, не більшими за k_1, а B_2 – підмножина B з елементами більшими k_1. Причому елемент k_1 після розбиття множини B буде перебувати на потрібному місці. Далі до множин B_1 і B_2 знову застосовують впорядкування швидким сортуванням.

Час роботи алгоритму швидкого сортування в гіршому випадку складає $O(n^2)$, але на практиці цей алгоритм виявляється одним із найшвидших.

```
double * quick(double *s,int low,int hi)
{
   double cnt,aux;
   int i,j;
   if (hi>low)
   {
      i=low;
      j=hi;
      cnt=s[i];
      while(i < j)
      {
         if (s[i+1]<=cnt)
          {
              s[i]=s[i+1];
              s[i+1]=cnt;
              i++;
          }
         else
```

```
            {
                if (s[j]<=cnt)
                {
                    aux=s[j];
                    s[j]=s[i+1];
                    s[i+1]=aux;
                }
                j--;
             }
          }
          quick(s,low,i-1);
          quick(s,i+1,hi);
       }
       return(s);
}
```

1.13 Структури

1.13.1 Оголошення структури

Структури дозволяють об'єднувати в єдиному об'єкті сукупність значень, які можуть мати різні типи. Оголошення структури здійснюється за допомогою ключового слова *struct*.

Синтаксис опису структури виглядає так :

```
struct [ім'я_структури]
{
  тип1 елемент1;
  тип2 елемент2;
  .....
  типN елементN;
} [список описів];
```

З метою ознайомлення з цим типом даних розглянемо найпростіший приклад представлення поняття "дата", що складається з декількох частин: число (день, місяць, рік), назва тижня та місяця:

```
struct date {
  int day ;
  int month ;
  int year;
  char day_name[15];
  char mon_name[14];
} arr[100],*pd,data,new_data;
```

В даному прикладі оголошуються:

data, new_data - змінні типу структури *date*;
pd – покажчик на тип *data*
arr – масив із 100 елементів, кожний елемент якого має тип *date*.

Можливий і наступний опис структури з використанням *typedef:*

```
typedef struct mystruct {
  int year;
  char size;
  float field;
} MYSTRUCT;
MYSTRUCT s;  /* те саме, що й struct mystruct s; */
```

Пам'ять розподіляється у структурі покомпонентно, зліва-направо, від молодших до старших адрес пам'яті (рис. 1.18).

```
typedef struct dataTypes {
   float aFloat;
   int anInt;
   char aString[8];
   char aChar;
   char aLong;
} DataTypes;

DataTypes data;
```

Рис. 1.18. Зберігання елементів структури у пам'яті

Потрібно відзначити, що на відміну від описів інших типів даних, опис структури не виділяє місця у пам'яті під елементи структури. Її опис визначає лише так званий шаблон, що описує характеристики змінних, що будуть розміщуватися у конкретній структурі. Щоб ввести змінні та зарезервувати для них пам'ять необхідно або після фігурної дужки, що завершує опис структури, вказати список ідентифікаторів,

як це зроблено у вищенаведеному прикладі, або окремо оголосити змінні типу, як ми це робимо у звичайних випадках.

Доступ до окремого елемента структури забезпечується операторами вибору: **.** (прямий селектор) та **->** (непрямий селектор), наприклад,

```
struct mystruct {
  int i;
  char str[21];
  double d;
} s,*sptr=&s;
s.i =3;
sptr->d = 1.23;
```

Ініціалізація структури подібна до тієї, що у масивах, але з урахуванням розміщення даних різного типу.

```
struct person {
  char frnm[20];
  char nm[30];
  int year;
  char s;
};

struct person poet={"Taras", "Shevtchenko",1814, 'M'},
       classics[]={{"Alfred", "Aho", 1939, 'M'},
                   {"Seimour", "Ginzburg",}, /* … */
                   {"Jeffrey", "Ulman", 1938, 'M'}};
```

У вищенаведеному прикладі ініціалізується змінна *poet* і масив структур *classics*. Значення classics[1].year і classics[1].s мають значення відповідно 0 і '\0'.

Для змінних одного і того ж самого структурного типу визначена операція присвоювання, при цьому здійснюється поелементне копіювання значень полів.

```
struct date {
  int day ;
  int month ;
  int year;
  char day_name[15];
  char mon_name[14];
} data,new_data;

/* ... */
data=new_data;
```

Але, для порівняння структур необхідно перевіряти рівність відповідних полів цих структур.

```
struct point
{
  float x,y;
  char c;
} point1,point2;
if (point1.x==point2.x&&point1.y==point2.y&&
                                    point1.c==point2.c)
{
   /* … */
};
```

Звертання до окремих елементів структури теж не викликає труднощів:

```
data.year=2005;
printf("%d-%d-%d",data.day,data.month,data.year);
scanf("%d",data.day);
gets(arr[0].day_name);
```

Доцільним та корисним є зв'язок структур та покажчиків, який дозволяє обійти деякі складні моменти. Так опис *date *pdate* утворить покажчик на структуру типу *date*. Використовуючи цей покажчик, можна звернутися до будь-якого елемента структури шляхом застосування операції -> , тобто *date ->year* , або що еквівалентно операції *(*pdate).year*. Однак слід зауважити, що спільне використання цих типів потребує від програміста достатньо високої кваліфікації, аби використовувати можливості найбільш ефективно та безпомилково.

Приклад 1.

```
#include<stdio.h>
#include<conio.h>
#define MAXTIT 41
#define MAXAUT 31

struct book
{
  char title[MAXTIT];
  char author[MAXAUT];
  float value;
};
```

```
void main()
{
  struct book libry;
  printf("Введіть назву книги.\n");
  gets(libry.title);

  printf("Тепер введіть прізвище автора.\n");
  gets(libry.author);
  printf("Тепер введіть ціну.\n");
  scanf("%f",&libry.value);
  printf("\n%s '%s',%g grn.\n",libry.author,
          libry.title,libry.value);
};
```

Кожний опис структури вводить унікальний тип структури, тому в наступному фрагменті програми:

```
    struct A {
       int i,j;
       double d;
    } a, a1;
    struct B {
       int i,j;
       double d;
    } b;
```

об'єкти a і a1 мають однаковий тип struct A, але об'єкти a і b мають різні типи структури. Структурам можна виконувати присвоювання тільки в тому випадку якщо і вихідна структура, і структура, які присвоюється мають один і той же тип.

```
a = a1; /*можна виконати, так як a і a1 мають однаковий тип */
a = b; /* помилка */
```

1.13.2 Масиви структур

Як і звичайними масивами простих типів, так само можна оперувати *масивами структур,* елементи якого мають структурований тип. Розглянемо наочний зразок, який ілюструє оголошення масиву структур:

```
typedef struct Date
{
  int d; /* день */
  int m; /* місяць */
  int y; /* рік */
} Date;
Date arr[100];
```

Вище було оголошено масив *arr*, що складається із 100 елементів, кожний з яких має тип *Data*. Кожний елемент масиву - це окрема змінна типу *Data*, що складається із трьох цілих елементів – *d*, *m*, *y*.

Доступ до полів структури аналогічний доступу до звичайних змінних, плюс використання індексу номеру елементу у квадратних дужках:

```
arr[25].d=24;
arr[12].m=12;
```

Запропонуємо програму, в якій реалізується концепція структурованого типу *Data*. Окремими функціями реалізуємо ініціалізацію елементів структури, додавання нового значення, виведення дати на екран, визначення високосного року.

```
#include<stdio.h>
#include<conio.h>
typedef struct Date
{
  int d; /* день */
  int m; /* місяць */
  int y; /* рік */
} Date;

void set_date_arr(Date *arr,Date value,int n)
{
 int i;
 for (i=0;i<n;i++)
 {
   arr[i].d=value.d;
   arr[i].m=value.m;
   arr[i].y=value.y;
 }
}

void print_date_arr(Date *arr,int n)
{
  int i;
  for (i=0;i<n;i++)
  {
    printf("%d.%d.%d\n",arr[i].d,arr[i].m,arr[i].y);
  }
}
```

```
void print_date(Date &d)
/* виведення на екран дати */
{
 printf("%d.%d.%d\n",d.d,d.m,d.y);
}


void init_date(Date &d,int dd,int mm,int yy)
/* ініціалізація структури типу Date */
{
  d.d=dd;
  d.m=mm;
  d.y=yy;
}

int leapyear(int yy)
/* визначення, чи високосний рік */ {
   if ((yy%4==0&&yy%100!=0)||(yy%400==0)) return 1;
     else return 0;
}

void add_year(Date &d,int yy)
/* додати yy років до дати */ {
  d.y+=yy;
}

void add_month(Date &d,int mm)
/* додати mm місяців до дати */ {
  d.m+=mm;
  if (d.m>12)
  {
    d.y+=d.m/12;
    d.m=d.m%12;
  }
}

void add_day(Date &d,int dd)
/* додати dd днів до дати */ {
 int days[]={31,28,31,30,31,30,31,31,30,31,30,31};
 d.d+=dd;
 if (leapyear(d.y)) days[1]=29;

 while ((d.d>days[d.m-1]))
 {
   if (leapyear(d.y)) days[1]=29;
      else days[1]=28;
   d.d-=days[d.m-1];
   d.m++;
```

```
    if (d.m>12)
    {
      d.y+=d.m%12;
      d.m=d.m/12;
    }
  }
}

void main(void)
{
  Date date1,date2;
  Date array[10]={{12,11,1980},{15,1,1982},{8,6,1985},
                  {8,8,1993},{20,12,2002},{10,1,2003}};
  clrscr();
  init_date(date1,15,12,2002);
  add_day(date1,16);
  print_date(date1);
  puts("");
  init_date(date2,1,1,2003);
  add_month(date2,10);
  print_date(date2);
  puts("");
  print_date_arr(array,6);
}
```

1.13.3 Бітові поля

Бітові поля (*bit fields*) – особливий вид полів структури. Вони дають можливість задавати кількість бітів, в яких зберігаються елементи цілих типів. Бітові поля дозволяють раціонально використовувати пам'ять за допомогою зберігання даних в мінімально потрібній кількості бітів.

При оголошенні бітового поля вслід за типом елемента ставиться двокрапка (:) і вказується цілочисельна константа, яка задає розмір поля (кількість бітів). Розмір поля повинен бути константою в діапазоні між 0 і заданим загальним числом бітів, яке використовується для зберігання даного типу даних.

```
struct  bit_field {
   int bit_1        : 1;
   int bits_2_to_5  : 4;
   int bit_6        : 1;
   int bits_7_to_16 : 10;
 } bit_var;
```

1.14 Об'єднання (union)

Об'єднання дозволяють в різні моменти часу зберігати в одному об'єкті значення різного типу. В процесі оголошення об'єднання з ним асоціюється набір типів, які можуть зберігатися в даному об'єднанні. В кожний момент часу об'єднання може зберігати значення тільки одного типу з набору. Контроль за тим, значення якого типу зберігається в даний момент в об'єднанні покладається на програміста.

Синтаксис :

```
union [ім'я_об'єднання]
{
  тип1 елемент1;
  тип2 елемент2;
  .....
  типN елементN;
} [список описів];
```

Пам'ять, яка виділяється під змінну типу об'єднання, визначається розміром найбільш довгого з елементів об'єднання. Всі елементи об'єднання розміщуються в одній і тій же області пам'яті з однієї й тієї ж адреси. Значення поточного елемента об'єднання втрачається, коли іншому елементу об'єднання присвоюється значення.

Приклад 1:

```
union sign
{
   int svar;
   unsigned uvar;
} number;
```

Приклад 2 :

```
union
{
   char *a,b;
   float f[20];
} var;
```

В першому прикладі оголошується змінна типу об'єднання з ім'ям *number*. Список оголошень елементів об'єднання містить дві змінні : *svar* типу *int* і *uvar* типу *unsigned*. Це об'єднання дозволяє запам'ятати ціле значення в знаковому або в без знаковому вигляді. Тип об'єднання має ім'я *sign*.

В другому прикладі оголошується змінна типу об'єднання з ім'ям *var*. Список оголошень елементів містить три оголошення : покажчика

a на значення типу *char*, змінної *b* типу *char* і масиву *f* з 20 елементів типу *float*. Тип об'єднання не має імені. Пам'ять, що виділяється під змінну *var*, рівна пам'яті, необхідної для зберігання масиву *f*, так як це найдовший елемент об'єднання.

1.15 Файлові потоки

В мові Сі та Сі++ файл розглядається як *потік* (*stream*), що представляє собою послідовність байтів, що записуються чи зчитуються. При цьому потік „не знає", що і в якій послідовності в нього записано. Розшифровка змісту написаних у ньому байтів лежить на програмі.

Таблиця 1.13. Значення аргументу mode функції fopen

"r"	відкриття файлу без дозволу на модифікацію, файл відкривається лише для читання.
"w"	створення нового файлу тільки для запису, якщо файл із вказаним ім'ям вже існує, то він перезапишеться.
"a"	відкриття файлу тільки для додавання інформації в кінець файлу, якщо файл не існує, він створюється.
"r+"	відкриття існуючого файлу для читання та запису.
"w+"	створення нового файлу для читання та запису, якщо файл із вказаним ім'ям вже існує, то він перезаписується.
"a+"	відкриває файл у режимі читання та запису для додавання нової інформації у кінець файлу; якщо файл не існує, він створюється.

Класичний підхід, прийнятий в Сі, полягає в тому, що інформація про потік заноситься в структуру *FILE*, яка визначена у файлі *stdio.h*. Файл відкривається за допомогою функції *fopen,* яка повертає покажчик на структуру типу *FILE*.

```
typedef struct
{
  short level;          /*рівень буферу */
  unsigned flags;       /*статус файлу */
  char fd;              /*дескриптор файлу */
  char hold;            /*попередній символ, якщо немає буферу */
  short bsize;           /*розмір буферу */
  unsigned char *buffer; /*буфер передавання даних */
  unsigned char *curp;    /*поточний активний покажчик */
  short token;            /*перевірка коректності */
} FILE;
```

Синтаксис функції *fopen* :

FILE *fopen(const char *filename, const char *mode);

Дана функція відкриває файл із заданим ім'ям і зв'язує з ним потік. Аргумент *mode* вказує режим відкриття файлу (таблиця 1.13).

До вказаних специфікаторів в кінці або перед символом „+" може додаватися символ „t" (текстовий файл), або „b" (бінарний, двійковий файл).

1.15.1 Текстові файли

Розглянемо спочатку роботу з текстовими файлами. Відкриття текстового файлу *test.txt* може мати вигляд :

```
#include<stdio.h>
void main()
{
  …
  FILE *f;
  if ((f=fopen("test.txt", "rt"))==NULL)
  {
    printf("Файл не вдалося відкрити.\n");
    return;
  }
  …
  fclose(f);
  …
}
```

В даному прикладі змінна *f* зв'язується з файлом *„test.txt"*, який відкривається як текстовий тільки для читання.

З відкритого таким чином файлу можна читати інформацію. Після закінчення роботи з файлом, його необхідно закрити за допомогою функції *fclose()*.

Якщо файл відкривався би за допомогою *fopen("test.txt", "rt+");* , то можна було б не тільки читати, але й записувати в нього інформацію.

З текстового файлу можна читати інформацію по рядках, по символах або за форматом.

Записування символу в файловий потік здійснюється функцією *putc*().

int putc(int ch, FILE *f);

Читання рядка здійснюється за допомогою функції *fgets()*.

char *fgets(char *s,int n,FILE *stream);

У виклику функції *fgets()* : *s* – покажчик на буфер, в який читається рядок, *n* – кількість символів. Читання символу в рядок проходить або до появи символу кінця рядка *„\n”*, або читається *n–1* символ. В кінці прочитаного рядка записується нульовий символ.

```
#include<stdio.h>
#include<string.h>
void main() {
  char s[80];
  FILE *f;
  if ((f=fopen("1.cpp", "rt"))==NULL)  {
     printf("There are an error\n");
     return;
  }
  do  {
     fgets(s,80,f);
     printf("%s",s);
  } while (!feof(f));
  fclose(f);
}
```

Функція *feof()* перевіряє, чи не прочитаний символ завершення файла. Якщо такий символ прочитаний, то *feof()* повертає ненульове значення і цикл завершується.

Читання з текстового файлу форматованих даних може здійснюватися функцією *fscanf*. Синтаксис :

int fscanf(FILE *stream, const char *format[, address, …]);

Параметр *format* визначає рядок форматування аргументів, які задаються своїми адресами.

При форматованому читанні можуть виникати помилки у зв’язку з досягненням завершення файлу або невірним форматом записаних у файлі даних. Перевірити, чи успішно пройшло читання даних можна за значенням, яке повертає функція *fscanf()*. При успішному читанні вона повертає кількість прочитаних полів. Тому читання даних можна організовувати наступним чином :

```
if (fscanf(f,”%d%d%d”,&a,&b,&c)!=3)
{
  printf(“Помилка читання!\n”);
};
```

Існує також і ряд функцій для запису даних у текстовий файл. Найчастіше використовуються функції *fgetc*(), *fputs()* та *fprintf()*.

Функція *fgetc*() використовується для читання чергового символу з потоку, відкритого функцією *fopen*().

int fgetc(FILE *f);

Синтаксис функції *fprintf()* :

int fprintf(FILE *stream, const char *format[,argument,…]);

Вона працює майже мак само, як і функція *printf(),* але їй потрібний додатковий аргумент для посилання на файл. Він є першим у списку аргументів. Наводимо приклад, який ілюструє звертання до наведених вище функцій:

```
#include<stdio.h>
void main()
{
  FILE *fi;
  int age;
  fi=fopen("age.txt","r"); /* відкриття файла для читання */
  fscanf(fi,"%d",&age); /*читання з файлу числового значення
*/
  fclose(fi);   /* закриття файла */
  fi=fopen("data.txt", "a"); /* відкриття файла для додавання
                  інформації в кінець */
  fprintf(fi, "Age==%d.\n",age); /* запис рядка в файл  */
  fclose(fi);   /* закриття файла */
}
```

1.15.2 Двійкові файли

Тепер розглянемо роботу з двійковими файлами. Двійковий файл представляє собою просто послідовність символів. Що саме і в якій послідовності зберігається в двійковому файлі – повинна знати програма.

Двійкові файли мають переваги, порівняно з текстовими при зберіганні числових даних. Операції читання і запису з такими файлами виконуються набагато швидше, ніж з текстовими, так як відсутня необхідність форматування (переведення в текстове представлення та навпаки). Двійкові файли зазвичай мають менший розмір, ніж аналогічні текстові файли. В двійкових файлах можна переміщуватися в будь-яку позицію і читати або записувати дані в

довільній послідовності, в той час, як в текстових файлах практично завжди виконується послідовна обробка інформації.

Про те, як відкривати двійкові файли було згадано раніше. Запис і читання в двійкових файлах виконується відповідно функціями *fwrite* і *fread*.

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

В обидві функції повинен передаватися покажчик *ptr* на дані, які вводяться або виводяться. Параметр *size* задає розмір в байтах даних, які читаються або записуються.

```
#include<stdio.h>
#include<conio.h>
struct mystruct {
  int i;
  char ch;
};

int main(void)
{
   FILE *stream;
   struct mystruct s;
   if ((stream = fopen("test.txt", "wb")) == NULL)
   {
      fprintf(stderr, "Неможливо відкрити файл\n");
      return 1;
   }
   s.i = 0;
   s.ch = 'A';
   fwrite(&s, sizeof(s), 1, stream);
   fclose(stream);
   return 0;
}
```

Тепер розглянемо особливості записування і читання рядків.

```
char s[10];
strcpy(s, "Example");
…
fwrite(s,strlen(s)+1,sizeof(char),stream);
```

Записування рядків відбувається посимвольно. В даному прикладі число символів, які записуються – *strlen(s)+1* (одиниця додається на нульовий символ в кінці). Читається рядок аналогічно:

```
fread(s,strlen(s)+1,sizeof(char),stream);
```

При цьому читання проходить теж посимвольно.

Дуже часто доводиться працювати з рядками різних довжин. В таких випадках можна перед рядком записати у файл ціле число, яке рівне числу символів у рядку.

```
…
int i=strlen(s)+1;
fwrite(&i,1,sizeof(int),stream);
fwrite(s,i,1,stream);
…
fread(&i,1,sizeof(int),stream);
fread(s,i,1,stream)
```

В усіх наведених вище прикладах читання даних проходило послідовно. Але, працюючи з двійковими файлами, можна організувати читання даних в довільному порядку. Для цього використовується „покажчик файла” (курсор), який визначає поточну позицію у файлі. При читанні даних курсор автоматично зміщується на число прочитаних байтів. Отримати поточну позицію курсору файла можна за допомогою функції *ftell()*.

long ftell(FILE *stream);

А встановлюється поточна позиція курсору у файлі за допомогою функції *fseek*():

int fseek(FILE *stream, long offset, int whence);

Ця функція задає зміщення на число байтів *offset* від точки відліку, яка визначається параметром *whence*. Цей параметр може приймати значення 0, 1, 2 (таблиця 1.14).

Таблиця 1.14. Можливі значення параметра whence функції fseek

Константа	*whence*	*Точка відліку*
SEEK_SET	0	Початок файлу
SEEK_CUR	1	Поточна позиція
SEEK_END	2	Кінець файлу

Якщо задане значення *whence=1*, то *offset* може приймати як додатне, так і від’ємне значення, тобто зсув вперед або назад.

Функція *rewind* переміщує курсор на початок файлу.

void rewind(FILE *stream);

Те ж саме можна зробити за допомогою функції *fseek()* :

```
fseek(stream, 0L, SEEK_SET);
```

Приклад програми, в якій використовуються описані вище функції :

```
#include <stdio.h>
long filesize(FILE *stream);
int main(void)
{
   FILE *stream;
   stream = fopen("test.txt", "w+");
   fprintf(stream, "This is a test");
   printf("Розмір файла test.txt рівний %ld байт\n",
              filesize(stream));
   fclose(stream);
   return 0;
}
long filesize(FILE *stream)
{
   long curpos, length;
   curpos = ftell(stream);
   fseek(stream, 0L, SEEK_END);
   length = ftell(stream);
   fseek(stream, curpos, SEEK_SET);
   return length;
}
```

1.15.3 Використання дескрипторів файлів

В мові Сі передбачений ще один механізм роботи з файлами – використання дескрипторів. Файли, які відкриваються таким чином не розраховані на роботу з буферами та форматованими даними.

На початку роботи будь-якої програми відкриваються п’ять стандартних потоків зі своїми дескрипторами.

Таблиця 1.15. Дескриптори стандартних потоків введення-виведення

потік	*дескриптор*	
stdin	0	стандартний вхідний потік
stdout	1	стандартний вихідний потік
stderr	2	стандартний потік повідомлень про помилки
stdaux	3	стандартний потік зовнішнього пристрою
stdprn	4	стандартний потік виведення на принтер

Але будь-яка програма може і явним чином відкривати будь-які файли з дескрипторами.

Функції, які працюють з дескрипторами файлів, описані в модулі *io.h*.

Файли відкривається функцією *open()*, яка повертає дескриптор файлу:

int open(const char *path, int access [, unsigned mode]);

Параметр *path* задає ім'я файлу відкриття. Параметр *access* визначає режим доступу до файлу. *mode* є не обов'язковим та задає режим відкриття файла.

Параметр *access* формується за допомогою операції АБО (|) з переліку прапорців.

O_RDONLY	тільки для читання
O_WRONLY	тільки для запису
O_RDWR	для читання і запису
O_CREAT	створення нового файлу
O_TRUNC	якщо файл існує, то він стає порожнім
O_BINARY	двійковий файл
O_TEXT	текстовий файл

Параметр *mode* може приймати наступні значення

S_IWRITE	дозволити запис
S_IREAD	дозволити читання

Використання функції *fopen()* демонструє наступний приклад :

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = open("TEST.TXT", O_CREAT | O_TEXT)) ==
-1)
   {
      perror("Error:");
      return 1;
   }
   write(handle, msg, strlen(msg));
   close(handle);
   return 0;
}
```

Як видно з прикладу, файл, відкритий функцією *open()* повинен бути закритий за допомогою функції *close()*.

```
int close(int handle);
```

Читання і запис даних при роботі з файлами, що визначаються дескрипторами *handle*, здійснюється функціями *write()* і *read()*.

```
int read(int handle, void *buf, unsigned len);
```

```
int write(int handle, void *buf, unsigned len);
```

В наведених функціях *buf* – покажчик на буфер, з якого записується в файл інформація, або в який читається *len* байтів з файла.

Буферизація потоків. В мові Сі існує ряд функцій, які дозволяють керувати буферизацією потоків.

Функція *setbuf*() дозволяє користувачу встановлювати буферизацію вказаного потоку *stream*. Синтаксис функції *setbuf*():

```
void setbuf(FILE *stream, char *buf);
```

Значення аргументу *stream* повинне відповідати стандартному або вже відкритому потоку.

Якщо значення аргументу *buffer* рівне *NULL*, то буферизацію буде відмінено. Інакше, значення аргументу *buffer* буде визначати адресу масиву символів довжини *BUFSIZ*, де *BUFSIZ* – розмір буфера (константа, визначена в *stdio.h*).

Визначений користувачем буфер використовується для буферизованого введення/виведення для вказаного потоку *stream* замість буферу, що виділяється системою по замовчуванню.

Потоки *stderr* і *stdout* по замовчуванню небуферизовані, але для них можна встановлювати буферизацію засобами *setbuf*.

Примітка. Наслідки буферизації будуть непередбаченими, якщо тільки функція *setbuf()* не викликана зразу вслід за функцією *fopen*() або *fseek*() для заданого потоку.

В мові Сі для керування буферизацією потоків існує ще одна функція: *setvbuf*(). Вона дозволяє користувачу керувати буферизацією та розміром буфера потоку *stream*. Синтаксис :

```
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Потік *stream* повинен відноситися до відкритого потоку.

Якщо значення параметру *buf* не *NULL,* то масив, адреса якого задається значенням параметра *buf* буде використовуватися в якості буфера.

Якщо потік буферизується, значення параметра *type* визначає тип буферизації. Тип буферизації може бути або *_IONBF*, або *_IOFBF*, або *_IOLBF*.

Якщо тип рівний *_IOFBF* або *_IOLBF*, то значення параметра *size* використовується як розмір буфера.

Якщо тип рівний *_IONBF*, то потік небуферизований, і значення параметрів *size* і *buf* ігноруються.

Допустиме значення параметра *size*: більше 0 і менше, ніж максимальний розмір цілого (*int*).

Значення констант *_IONBF*, *_IOFBF* та *_IOLBF* визначені у файлі *stdio.h.*

```
_IOFBF  0  /* буферизація на повний об'єм буфера */
_IOLBF  1  /* порядкова буферизація */
_IONBF  2  /* потік не буферизується */
```

Для примусового виштовхування буферу можна використовувати функцію *fflush*(). Її синтаксис :

```
int fflush(FILE *stream);
```

Дана функція виштовхує вміст буфера, зв'язаного з потоком *stream*. Потік залишається відкритим. Якщо потік небуферизований, то виклик функції *fflush*() не викличе ніяких ефектів.

Буфер потоку автоматично виштовхується, коли він заповнюється, коли закривається потік або коли програма завершує своє виконання.

Приклад 1.

```
#include <stdio.h>
#include<conio.h>
char outbuf[BUFSIZ];
int main(void)
{
   clrscr();
   setbuf(stdout, outbuf);
   puts("This is a test of buffered output.\n\n");
   puts("This output will go into outbuf\n");
   puts("and won't appear until the buffer\n");
   puts("fills up or we flush the stream.\n");
   getch();
   fflush(stdout);
```

```
   getch();
   return 0;
}
```

Приклад 2.

```
#include <stdio.h>
int main(void)
{
   FILE *input, *output;
   char bufr[512];
   input = fopen("file.in", "r+b");
   output = fopen("file.out", "w");
   if (setvbuf(input, bufr, _IOFBF, 512) != 0)
        printf("Помилка встановлення буферизації\
                      для вхідного файла\n");
   else
      printf("Для вхідного файла встановлено \
буферизацію\n");
   if (setvbuf(output, NULL, _IOLBF, 132) != 0)
      printf("Помилка встановлення буферизації \
                для вихідного файла\n");
   else
      printf("Буфер для вихідного файла \
встановлено\n");
   fclose(input);
   fclose(output);
   return 0;

}
```

1.16 Функціональний підхід

Як визначити термін "програма"? Взагалі це послідовність операцій над структурами даних, що реалізують алгоритм розв'язання конкретної задачі. На початку проектування задачі ми розмірковуємо відносно того, що повинна робити наша програма, які конкретні задачі вона повинна розв'язувати, та які алгоритми при цьому повинні бути реалізовані. Буває, і це характерно для більшості задач, вихідна задача досить довга та складна, у зв'язку з чим програму складно проектувати та реалізовувати, а тим більше супроводжувати, якщо не використовувати методів керування її розмірами та складністю. Для цього потрібно використати відомі прийоми функціонально-модульного програмування для структурування програм, що полегшує їх створення, розуміння суті та супровід.

Розв'язання практичної задачі проходить у кілька етапів, зміст яких подає таблиця 1.16.

Організація програми на Сі досить логічна. Мова Сі надає надзвичайно високу гнучкість для фізичної організації програми. Нижче наведена типова організація невеликого програмного проекту на Сі:

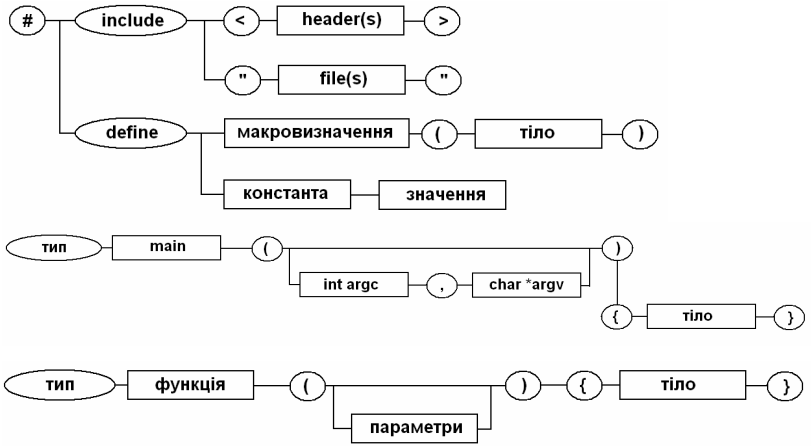


Рис. 1.19. Типова структура програм на Сі

Нижче піде мова про процедурне (функціональне) програмування на *Сі*. Існують досить добре розвинуті методи процедурного програмування, що базуються на моделі побудови програми як деякої сукупності функцій. Прийоми програмування пояснюють, як розробляти, організовувати та реалізовувати функції, що складають програму.

Структура кожної функції співпадає зі структурою головної функції програми *main()*. Функції іноді ще називають підпрограмами.

Основу процедурного програмування на будь-якій мові програмування складає процедура (походить від назви) або функція (як різновид, що саме відповідає мові програмування *Сі*).

Функція - модуль, що містить деяку послідовність операцій. Її розробка та реалізація у програмі може розглядатися як побудова операцій, що вирішують конкретну задачу (підзадачу). Однак взагалі функція може розглядатися окремо як єдина абстрактна операція, і, щоб її використовувати, користувачеві необхідно зрозуміти інтерфейс функції - її вхідні дані та результати виконання. Легко буде зрозуміти

ту функцію, що відповідає абстрактним операціям, необхідним для рішення задачі. Функцію та її використання у програмі можна у такому разі представляти у термінах задачі, а не в деталях реалізації. Припустимо, необхідно розробити функціональний модуль, що розв'язує наступне завдання: існує вхідний список певних даних, який необхідно відсортувати, переставляючи його елементи у визначеному порядку. Ця функція може бути описана, як абстрактна операція сортування даних, що може бути частиною вирішення деякої підмножини задач. Функція, що реалізує цю операцію, може бути використана у багатьох програмах, якщо вона створена як абстракція, що не залежить від реалізації (контексту програми).

Таблиця 1.16. Типові етапи розв'язання задач

Етапи	Опис
Постановка задачі та її змістовний аналіз	1. Визначити вхідні дані, які результати необхідно отримати і в якому вигляді подавати відповіді. 2. Визначити за яких умов можливо отримати розв'язок задачі, а за яких - ні. 3. Визначити, які результати вважатимуться вірними.
Формалізація задачі, вибір методу її розв'язання. (математичне моделювання задачі)	1. Записати умову задачі за допомогою формул, графіків, рівнянь, нерівностей, таблиць тощо. 2. Скласти математичну модель задачі, тобто визначити зв'язок вихідних даних із відповідними вхідними даними за допомогою математичних співвідношень з урахуванням існуючих обмежень на вхідні, проміжні та вихідні дані, одиниці її виміру, діапазон зміни тощо. 3. Вибрати метод розв'язку задачі.
Складання алгоритму розв'язання задачі	Алгоритм більшою мірою визначається обраним методом, хоча один і той самий метод може бути реалізований за допомогою різних алгоритмів. Під час складання алгоритму необхідно враховувати всі його властивості.
Складання програми	Написання програми на мові програмування
Тестування і відлагодження програми	Перевірка правильності роботи програми за допомогою тестів і виправлення наявних помилок. Тест - це спеціально підібрані вхідні дані та результати, отримані в результаті обробки програмою цих даних.
Остаточне виконання програми, аналіз результатів	Після остаточного виконання програми необхідно провести аналіз результатів. Можлива зміна самого підходу до розв'язання задачі та повернення до першого етапу для повторного виконання усіх етапів.

Функції мають параметри, тому їх операції узагальнені для використання будь-якими фактичними аргументами відповідного типу. Що є вхідними даними для функції ? Вхідними даними для неї є аргументи та глобальні структури даних, що використовуються функцією. Вихідними даними є ті значення, які функція повертає, а також зміни глобальних даних, модифікації.

Функціональний модуль, що не використовує глобальні дані, параметризується вхідними параметрами. Функція – це операція над будь-якими аргументами відповідного типу, адже вона не оперує конкретними об'єктами у програмі. Тому її можна використовувати безліч разів з різними параметрами, і не тільки в одній програмі, а й в інших із структурами даних того ж типу. Інтерфейс буде зрозумілий з опису прототипу функції, а об'єкти даних, описані в його реалізації, зрозумілі з локальних оголошень функції. Тому при параметризації входу та локалізації описів функція представляє собою тип самодокументованого модуля, який легко використовувати. Крім цього, функціям притаманна модульність. Її широко використовують для надання функціям більшої ясності, можливості повторного використання, що, таким чином, допомагає скоротити витрати, пов'язані з її реалізацією та супроводом.

1.16.1 Функції

Як було сказано вище, функції можуть приймати параметри і повертати значення. Будь-яка програма на мові Сі складається з функцій, причому одна з яких обов'язково повинна мати ім'я *main()*.

Синтаксис опису функції має наступний вигляд :

```
тип_поверт_значення ім'я_функції ([список_аргументів])
{
    оператори тіла функції
}
```

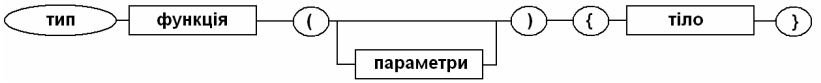


Рис. 1.20. Синтаксис опису функції

Слід чітко розрізняти поняття опису та представлення функцій.

Опис функції задає її ім'я, тип значення, що повертається та список параметрів. Він дає можливість організувати доступ до функції

(розташовує її в область видимості), про яку відомо, що вона *external* (зовнішня). Представлення визначає, задає дії, що виконуються функцією при її активізації (виклику).

Оголошенню функції можуть передувати специфікатори класу пам'яті *extern* або *static*.

- *extern* – глобальна видимість у всіх модулях (по замовчуванню);
- *static* – видимість тільки в межах модуля, в якому визначена функція.

Тип значення, яке повертається функцією може бути будь-яким, за виключенням масиву та функції (але може бути покажчиком на масив чи функцію). Якщо функція не повертає значення, то вказується тип *void*.

1.16.2 Функції, що не повертають значення

Функції типу *void* (ті, що не повертають значення), подібні до процедур Паскаля. Вони можуть розглядатися як деякий різновид команд, реалізований особливими програмними операторами. Оператор *func();* виконує функцію *void func()* , тобто передасть керування функції, доки не виконаються усі її оператори. Коли функція поверне керування в основну програму, тобто завершить свою роботу, програма продовжить своє виконання з того місця, де розташовується наступний оператор за оператором *func()*.

```
/*демонстраційна програма*/
#include<stdio.h>
void func1(void);
void func2(void);
main()
{
  func1();
  func2();
  return 0;
}
void func1(void)
{
  /* тіло */
}
void func2(void)
{
  /* тіло */
}
```

Звернемо увагу на те, що текст програми починається з оголошення прототипів функцій - схематичних записів, що повідомляють компілятору ім'я та форму кожної функції у програмі. Для чого використовуються прототипи? У великих програмах це правило примушує Вас планувати проекти функцій та реалізовувати їх таким чином, як вони були сплановані. Будь-яка невідповідність між прототипом (оголошенням) функції та її визначенням (заголовком) призведе до помилки компіляції. Кожна з оголошених функцій має бути визначена у програмі, тобто заповнена операторами, що її виконують. Спочатку йтиме заголовок функції, який повністю співпадає з оголошеним раніше прототипом функції, але без заключної крапки з комою. Фігурні дужки обмежують тіло функції. В середині функцій можливий виклик будь-яких інших функцій, але неможливо оголосити функцію в середині тіла іншої функції. Нагадаємо, що Паскаль дозволяє працювати із вкладеними процедурами та функціями.

Надалі розглянемо приклад програми, що розв'язує відоме тривіальне завдання - обчислює корені звичайного квадратного рівняння, проте із застосуванням функціонального підходу:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
float A,B,C;
/*функція прийому даних*/
void GetData()
{
  clrscr();
  printf("Input A,B,C:");
  scanf("%f%f%f",&A,&B,&C);
}
/*функція запуску основних обчислень*/

void Run()
{
  float D;
  float X1, X2;
  if ((A==0) && (B!=0))
  {
    X1 = (-C)/B;
    printf("\nRoot: %f",X1);
    exit(0);
  }
  D = B*B - 4*A*C;
```

```
  if (D<0) printf("\nNo roots...");
  if (D==0)
  {
    X1=(-B)/(2*A);
    printf("\nTwo equal roots: X1=X2=%f",X1);
  }
  if (D>0)
  {
    X1 = (-B+sqrt(D))/(2*A);
    X2 = (-B-sqrt(D))/(2*A);
    printf("\nRoot X1: %f\nRoot X2: %f",X1,X2);
  }
}

/*головна функція програми/
void main()
{
   GetData();
   Run();
}
```

Якщо в описі функції не вказується її тип, то по замовчуванню він приймається як тип *int.* У даному випадку обидві функції описані як *void*, що не повертають значення**.** Якщо ж вказано, що функція повертає значення типу *void*, то її виклик слід організовувати таким чином, аби значення, що повертається, не використовувалося б.

Просто кажучи, таку функцію неможливо використовувати у правій частині виразу. В якості результату функції остання не може повертати масив, але може повертати покажчик на масив. У тілі будь-якої функції може бути присутнім вираз *return;* який не повертає значення. І, насамкінець, усі програмні системи, написані за допомогою мови *Ci* , повинні містити функцію *main(),* що є вхідною точкою будь-якої системи. Якщо вона буде відсутня, завантажувач не зможе зібрати програму, про що буде отримано відповідне повідомлення.

1.16.3 Передача параметрів

Усі параметри, за винятком параметрів типу покажчик та масивів, передаються за значенням. Це означає, що при виклику функції їй передаються тільки значення змінних. Сама функція не в змозі змінити цих значень у викликаючій функції. Наступний приклад це демонструє:

```
#include<stdio.h>
void test(int a) {
  a=15;
  printf("    in test : a==%d\n",a);
};
void main() {
  int a=10;
  printf("before test : a==%d\n",a);
  test(a);
  printf("after test  : a==%d\n",a);
};
```

При передачі параметрів за значенням у функції утворюється локальна копія, що приводить до збільшення об'єму необхідної пам'яті. При виклику функції стек відводить пам'ять для локальних копій параметрів, а при виході з функції ця пам'ять звільняється. Цей спосіб використання пам'яті не тільки потребує додаткового її об'єму, але й віднімає додатковий час для зчитування. Наступний приклад демонструє, що при активізації (виклику) функції копії створюються для параметрів, що передаються за значенням, а для параметрів, що передаються за допомогою покажчиків цього не відбувається. У функції два параметри - *one*, *two* - передаються за значенням, *three* - передається за допомогою покажчика. Так як третім параметром є покажчик на тип int, то він, як і всі параметри подібного типу, передаватиметься за вказівником:

```
#include <stdio.h>
void test(int one, int two, int * three)
{
  printf( "\nАдреса one дорівнює %p", &one );
  printf( "\nАдреса two дорівнює %p", &two );
  printf( "\nАдреса three дорівнює %p", &three );
  *three+=1;
}
main()
{
  int a1,b1;
  int c1=42;
  printf( "\nАдреса a1 дорівнює %p", &a1 );
  printf( "\nАдреса b1 дорівнює %p", &b1 );
  printf( "\nАдреса c1 дорівнює %p", &c1 );
  test(a1,b1,&c1);
  printf("\nЗначення c1 = %d\n",c1);
}
```

На виході ми отримуємо наступне:

```
Адреса a1 дорівнює FEC6
Адреса b1 дорівнює FEC8
Адреса c1 дорівнює FECA
Адреса one дорівнює FEC6
Адреса two дорівнює FEC8
Адреса three дорівнює FECA
Значення c1 = 43
```

Після того, як змінна **tree* в тілі функції *test* збільшується на одиницю, нове значення буде присвоєно змінній *c1,* пам'ять під яку відводиться у функції *main().*

У наступному прикладі напишемо програму, що відшукує та видаляє коментарі з програми на *Ci*. При цьому слід не забувати коректно опрацьовувати рядки у лапках та символьні константи. Вважається, що на вході - звичайна програма на *Ci*. Перш за все напишемо функцію, що відшукує початок коментарю (/*):

```
/*функція шукає початок коментарю */
void rcomment(int c) {
  int d;
  if (c=='/')
    if (( d=getchar())=='*')
      in_comment();
    else if (d=='/') {
        putchar(c);
        rcomment(d);
      }
         else {
            putchar(c);
            putchar(d);
         }
  else if (c=='\''|| c=='"') echo_quote(c);
       else
         putchar(c);
}
```

Функція *rcomment(int c)* відшукує початок коментарю, а коли знаходить, викликає функцію *in_comment*(), що відшукує кінець коментарю. Таким чином, гарантується, що перша процедура дійсно ігноруватиме коментар:

```
/*функція відшукує кінець коментарю */
void in_comment(void)
{
  int c,d;
  c=getchar();
  d=getchar();
  while (c!='*'|| d!='/')
  {
    c=d;
    d=getchar();
  }
}
```

Крім того, функція *rcomment(int c)* шукає також одинарні та подвійні дужки, та якщо знаходить, викликає *echo_quote(int c)*. Аргумент цієї функції показує, зустрілась одинарна або подвійна дужка. Функція гарантує, що інформація всередині дужок відображається точно та не приймається помилково за коментар:

```
/*функція відображає інформацію без коментарю */
void echo_quote(int c)
{
   int d;
   putchar(c);
   while ((d=getchar()) !=c)
   {
      putchar(d);
      if (d=='\\')
        putchar(getchar());
   }
   putchar(d);
}
```

До речі, функція *echo_quote(int c)* не вважає лапки, що слідують за зворотною похилою рискою, заключними. Будь-який інший символ друкується так, як він є насправді. А на кінець текст функції *main*() даної програми, що відкривається переліком прототипів визначених нами функцій:

```
/* головна  програма */
#include <stdio.h>
void rcomment(int c);
void in_comment(void);
void echo_quote(int c);
```

```
main()
{
  int c,d;
  while ((c=getchar())!=EOF)
    rcomment(c) ;
  return 0;
}
```

Програма завершується, коли *getchar()* повертає символ кінця файлу. Це був типовий випадок проектування програми із застосуванням функціонального підходу.

1.16.4 Функції із змінним числом параметрів

Інколи у функції потрібно передати деяке число фіксованих параметрів та невизначене число додаткових. В цьому випадку опис функції буде мати вигляд :

тип ім'я_функції(список параметрів, ...)

Список аргументів включає в себе скінченне число обов'язкових параметрів (цей список не може бути порожнім), після якого на місці невизначеного числа параметрів ставиться три крапки. Для роботи з цими параметрами у файлі stdarg/h визначений тип списку *va_list* і три макроси: *va_start*, *va_arg*, *va_end*.

Макрос *va_start* має синтаксис :

void va_start(va_list ap, lastfix)

Цей макрос починає роботу зі списком, встановлюючи його покажчик *ap* на перший аргумент зі списку аргументів з невизначеним числом.

Макрос *va_arg* має синтаксис :

void va_arg(va_list ap, type)

Цей макрос повертає значення наступного (чергового) аргументу зі списку. Перед викликом *va_arg* значення *ap* повинне бути встановлене викликом *va_start* або *va_arg*. Кожний виклик *va_arg* переводить покажчик на наступний аргумент.

Макрос *va_end* має синтаксис :

void va_end(va_list ap)

Даний макрос завершує роботу зі списком, звільняючи пам'ять.

```
#include <stdio.h>
#include <stdarg.h>

void sum(char *msg, ...)
{
   int total = 0;
   va_list ap;
   int arg;
   va_start(ap, msg);
   while ((arg = va_arg(ap,int)) != 0)
   {
      total += arg;
   }
   printf(msg, total);
   va_end(ap);
}

int main(void)
{
   sum("Сума 1+2+3+4 рівна %d\n", 1,2,3,4,0);
   return 0;
}
```

1.16.5 Рекурсивні функції

Рекурсія – це спосіб організації обчислювального процесу, при якому функція в ході виконання операторів звертається сама до себе.

Функція називається *рекурсивною*, якщо під час її виконання можливий повторний її виклик безпосередньо (прямий виклик) або шляхом виклику іншої функції, в якій міститься звертання до неї (непрямий виклик).

Прямою (безпосередньою) *рекурсією* називається рекурсія, при якій всередині тіла деякої функції міститься виклик тієї ж функції.

```
void fn(int i) {
   /* ... */
   fn(i);
   /* ... */
}
```

Непрямою рекурсією називається рекурсія, що здійснює рекурсивний виклик функції шляхом ланцюга викликів інших функцій. При цьому всі функції ланцюга, що здійснюють рекурсію, вважаються також рекурсивними.

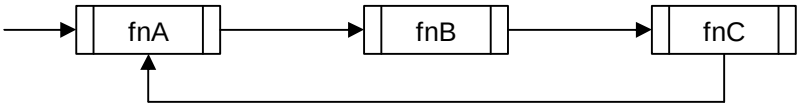


Рис. 1.21. Непряма рекурсія

```
void fnA(int i);
void fnB(int i);
void fnC(int i);
void fnA(int i) {
   /* ... */
   fnB(i);
   /* ... */
}
void fnB(int i) {
   /* ... */
   fnC(i);
   /* ... */
}
void fnC(int i) {
   /* ... */
   fnA(i);
   /* ... */
}
```

Якщо функція викликає сама себе, то в стеку створюється копія значень її параметрів, як і при виклику звичайної функції, після чого управління передається першому оператору функції. При повторному виклику цей процес повторюється.

В якості прикладу розглянемо функцію, що рекурсивно обчислює факторіал. Як відомо, значення факторіала обчислюється за формулою: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$, причому $1! = 1$ і $0! = 1$. Факторіал також можна обчислити за допомогою простого рекурентного співвідношення $n! = n \cdot (n-1)!$. Для ілюстрації рекурсії скористаємося саме цим співвідношенням.

```
#include<stdio.h>
#include<conio.h>
double fact(int n)
{
   if (n<=1) return 1;
   return (fact(n-1)*n);
}
void main()
```

```
{
   int n;
   double value;
   clrscr();
   printf("N=");
   scanf("%d",&n);
   value=fact(n);
   printf("%d! = %.50g",n,value);
   getch();
}
```

Роботу рекурсивної функції *fact*() розглянемо на прикладі n=6! За рекурентним співвідношенням : $6! = 5! \cdot 6$. Таким чином, щоб обчислити 6! ми спочатку повинні обчислити 5!. Використовуючи співвідношення, маємо, що $5! = 4! \cdot 5$, тобто необхідно визначити 4!. Продовжуючи процес, отримаємо :

1). $6! = 5! \cdot 6$ 2). $5! = 4! \cdot 5$ 3). $4! = 3! \cdot 4$ 4). $3! = 2! \cdot 3$ 5). $2! = 1! \cdot 2$ 6). $1! = 1$

В кроках 1-5 завершення обчислення кожний раз відкладається, а шостий крок є ключовим. Отримане значення, яке визначається безпосередньо, а не як факторіал іншого числа. Відповідно, ми можемо повернутися від 6-ого кроку до 1-ого, послідовно використовуючи значення :

6).1!=1 5).2!=2 4). 3!=6 3). 4!=24 2). 5!=120 1). 6!=720

Важливим для розуміння ідеї рекурсії є те, що в рекурсивних функціях можна виділити дві серії кроків.

Перша серія – це *кроки рекурсивного занурення функції* в саму себе до тих пір, поки вибраний параметр не досягне граничного значення. Ця важлива вимога завжди повинна виконуватися, щоб функція не створила нескінченну послідовність викликів самої себе. Кількість таких кроків називається глибиною рекурсії.

Друга серія – це *кроки рекурсивного виходу* до тих пір, поки вибраний параметр не досягне початкового значення. Вона, як правило забезпечує отримання проміжних і кінцевих результатів.

1.16.6 Покажчики на функції

Як згадувалося раніше, на функцію, як і на інший об’єкт мови Сі можна створити покажчик.

```
float (*func)(float a, float b);
```
/* покажчик на функцію, що приймає два параметри типу float і повертає значення типу float */

Покажчики на функції широко використовується для передачі функцій як параметрів іншим функціям.

За означенням покажчик на функцію містить адресу першого байта або слова виконуваного коду функції. Над покажчиками на функцію заборонені арифметичні операції.

Розглянемо приклад, що містить грубу помилку, спробу працювати з непроініціалізованим покажчиком.

```
#include<stdio.h>
#include<conio.h>
void main(void)
{
  void (*efct)(char *s);
```
/* змінний-покажчику виділена ОП, але efct не містить значення адреси ОП для функції */
```
  efct("Error");
```
/* груба помилка – спроба працювати з неініціалізованим покажчиком*/
```
}
```

Для того, щоб можна було використовувати покажчик на функцію потрібно спочатку присвоїти йому значення адреси пам'яті, де розташована функція, як це зроблено в наступному прикладі.

```
#include<stdio.h>
#include<conio.h>
void print(char *s)
{
   puts(s);
}

void main(void)
{
   void (*efct)(char *s);
   efct=&print;                    /* efct=print */
   (*efct)("Function Print!");     /* efct("Function Print!"); */
}
```

Для отримання значення адреси функції необов'язково використовувати операцію &. Тому наступні присвоювання будуть мати однаковий результат :

```
1). efct=&print;
2). efct=print;
```

Операція розіменування покажчика на функцію * також є необов'язковою.

```
1). (*efct)("Function Print!");
2). efct("Function Print!");
```

Покажчикам на функції можна присвоювати адреси стандартних бібліотечних функцій.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

void main(void)
{
  double (*fn)(double x);
  float y,x=1;
  fn=sin;
  y=fn(x);
  printf("sin(%g)==%g\n",x,y);
  fn=cos;
  y=fn(x);
  printf("cos(%g)==%g\n",x,y);
}
```

Покажчики на функції можуть також виступати в якості аргументів функцій.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

double fn(double (*pfn)(double x),double x)
{
   double y=pfn(x);
   printf("y==%g\n",y);
   return y;
}

double sin_cos(double x)
{
   return sin(x)*cos(x);
}
```

```
void main(void)
{
   fn(sin,1);
   fn(&cos,1);
   fn(&sin_cos,1);
}
```

1.16.7 Класи пам'яті

Будь-яка змінна та функція, описана у програмі на Сі, належить до конкретного класу пам'яті, що визначає час її існування та область видимості. *Час існування змінної* – це період, протягом якого змінна існує в пам'яті, а *область видимості* (область дії) – це частина програми, в якій змінна може використовуватися.

В мові Сі існує чотири специфікатори класу пам'яті: *auto*, *register*, *extern* і *static*.

Таблиця 1.17. Область дії та час існування змінних різних класів пам'яті

Клас пам'яті	*Ключове слово*	*Час існування*	*Область дії*
Автоматичний	auto	тимчасово	блок
Регістровий	register	тимчасово	блок
Статичний локальний	static	постійно	блок
Статичний глобальний	static	постійно	файл
Зовнішній	extern	постійно	програма

Клас пам'яті для функції завжди *external*, якщо перед її описом не стоїть специфікатор *static*. Клас пам'яті конкретної змінної залежить або від місця розташування її опису, або задається явно за допомогою спеціального специфікатору класу пам'яті, що розташовується перед описом функції. Усі змінні *Сі* можна віднести до одного з наступних класів пам'яті:

1) auto (автоматична, локальна)

Ключове слово *auto* використовується рідко. Кожна змінна, описана в тілі функції (в середині блоку), обмеженого фігурними дужками, відноситься до класу пам'яті автоматичних (локальних) змінних:

```
int anyfunc(void)
{
  char item;
  ........
}
```

Область дії локальної змінної *item* поширюється лише на блок, в якому вона оголошена. Пам'ять відводиться під змінну динамічно, під час виконання програми при вході у блок, в якому описана відповідна змінна. Локальна змінна тимчасово зберігається в стеку, коли функція починає свою роботу. Після закінчення роботи функції, або при виході з блоку знищує виділену стекову пам'ять, відкидаючи за необхідністю всі збережені змінні, тобто при виході з блоку пам'ять, відведена під усі його автоматичні змінні, автоматично звільняється (звідси й термін – *automatic*). З цієї причини декілька функцій безконфліктно можуть оголошувати локальні змінні з ідентичними іменами (це найчастіше буває з іменами лічильників циклів, індексів масивів тощо).

Отже, область видимості такої змінної розпочинається з місця її опису і закінчується в кінці блоку, в якому змінна описана. Доступ до таких змінних із зовнішнього блоку неможливий.

Застосування автоматичних змінних в локальних блоках дозволяє наближати опис таких змінних до місця їх розташування. Наступний приклад демонструє опис автоматичних змінних в середині блоку:

```
#include <stdio.h>
void main()
{
  printf("\n Знаходимося в main().");
  {
    int i;
    for(i=10;i>0;i--)
       printf("\n%d",i);
    printf("\n");
  }
}
```

2) register (регістрова)

Цей специфікатор може використовуватися лише для автоматичних змінних або для формальних параметрів функції. Він вказує компілятору на те, що користувач бажає розмістити змінну не в оперативній пам'яті, а на одному з швидкодіючих регістрів комп'ютеру, від чого програма виконуватиметься більш ефективніше.

Звісно, це стосується перш за все саме тих змінних, звертання до яких у функції виконуватиметься найчастіше. На практиці на цей тип змінних накладаються деякі обмеження, що відображають реальні можливості конкретної машини. У випадку надлишкових та недопустимих описів подібний специфікатор просто ігнорується.

3) extern (зовнішня, глобальна)

Будь-яка змінна, описана не в тілі функції (без специфікатору класу пам'яті), по замовчуванню відноситься до *extern* - змінних (або глобальних змінних). Глобальні змінні продовжують існувати протягом усього життєвого циклу програми. Якщо користувач не вкаже ініційоване значення таким змінним, їм буде присвоєно початкове нульове значення. Найчастіше оголошення таких змінних розташовується безпосередньо перед *main*():

```
/*file1.c*/
#include <stdio.h>
int globalvar;
main()
{
   /* operators */
}
```

Будь-які оператори у будь-якій функції файлу file1.c можуть виконувати читання та запис змінної *globalvar*. Але це ще не все! Виявляється, що глобальні змінні завжди залишаються під контролем завантажувача програми, що здійснює збірку програми із множини obj-файлів. Саме завдяки цьому до зовнішніх змінних можливий доступ з інших файлів. Для того, аби таку змінну можна було б використовувати в іншому файлі, слід задати специфікатор *extern*:

```
/*file2.c*/
#include<stdio.h>
void main()
{
   extern globalvar;
   printf("globalvar : %d", globalvar);
)
```

Опис *extern globalvar*; вказує компілятору на те, що ця змінна визначена як зовнішня та її опис знаходиться за межами даного файлу. У даному випадку опис *extern* розташований в середині функції, тому його дія впливає тільки на дану функцію. Якщо розмістити його ззовні будь-якої функції, то його дія поширюється на весь файл від точки опису.

Цікаво, якщо в середині блоку описана автоматична змінна, ім'я якої співпадає з іменем глобальної змінної, то в середині блоку глобальна змінна маскується локальною. Це означає, що в такому блоці видною буде саме автоматична, тобто локальна змінна.

4) static (статична)

Щоб обмежити доступ до змінних, дозволяючи зберегти їх значення між викриками функцій, слід оголошувати їх статичними. Статична змінна може бути *внутрішньою* або *зовнішньою*. Внутрішні статичні змінні локальні по відношенню до окремої функції, подібно автоматичним, проте на відміну від останніх продовжують існувати, а не виникають та знищуються при кожній активації функції. Це означає, що внутрішні статичні змінні є власною, постійною пам'яттю для функції:

```
int funct(void)
{
  static int value=20;
  ...
}
```

Компілятор відведе постійну область пам'яті для змінної *value* та проініціалізує її значення. Ця ініціалізація не повторюватиметься щоразу при активації функції. В подальшому змінна матиме те значення, яке вона отримала по завершенні її останньої роботи. Слід відзначити, що така змінна буде назавжди прихованою для завантажувача даної програми. Тому область дії статичних змінних обмежена функцією, в якій вона була оголошена, а функції не мають доступу до статичних змінних, оголошених в інших функціях.

Зовнішні статичні об'єкти відомі в тому файлі, в якому описані, проте в інших файлах вони невідомі. Таким чином, забезпечується спосіб об'єднання даних та маніпулюючи ними підпрограм таким чином, що інші підпрограми та дані у будь-якому випадку не зможуть конфліктувати з ними.

1.16.8 Додаткові можливості функції main()

Потрібно зауважити, що функція *main()* може як повертати деяке значення в операційну систему, так і приймати параметри.

тип main(int argc, char* argv[], char *env[]) { /* … */ }

Імена параметрів можуть мати будь-які назви, але прийнято використовувати *argc, argv* та *env*. Перший параметр *argc* містить ціле число аргументів командного рядка, що посилається функції *main()*;

argv – це масив покажчиків на рядки. Для версії ДОС *argv[0]* містить повний шлях програми, що в даний момент виконується, *argv[1]* та *argv[2]* відповідно вказує на перший та другий після імені програми параметри командного рядка, *argv[argc-1]* вказує на останній аргумент, *argv[argc]* містить *NULL*.

env – це масив покажчиків на рядки, причому кожний елемент *env[]* містить рядок типу *ENVVAR=значення*. *ENVVAR* - це ім'я змінної середовища.

Можливо для першого ознайомлення з Сі ця інформація не є обов'язковою, проте не може не зацікавити приклад програми, що демонструє найпростіший шлях використання аргументів, що передаються функції *main*():

```
/* Використання аргументів функції main() */
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[], char *env[])
{
  int i;
  printf("Значення argc = %d \n\n",argc);
  printf("В командному рядку міститься %d параметрів \n",argc);
  for (i=0; i<argc; i++)
    printf(" argv[%d]: %s\n", i, argv[i]);
  printf("Середовище містить наступні рядки:\n");
  for (i=0; env[i] != NULL; i++)
    printf(" env[%d]: %s\n", i, env[i]);
}
```

Організуємо виконання програми з командним рядком таким чином:

```
C:> c:\tc\testargs.exe 1_st_arg "2_arg " 3 4 "dummy" stop!
```

В результаті роботи програми ви отримаєте приблизно наступне:

```
Значення argc = 7
В командному рядку міститься 7 параметрів
argv[0]: c:\tc\testargs.exe
argv[1]: 1_st_arg
argv[2]: 2_arg
argv[3]: 3
argv[4]: 4
argv[5]: dummy
argv[6]: stop!
Середовище містить наступні рядки:
env[0]: COMSPEC=C:\COMMAND.COM
env[1]: PROMPT=$p $g
env[2]: PATH=C:\SPRINT;C:\DOS;C:\TC
```

Максимальна загальна довжина командного рядка, включаючи пробіли та ім'я самої програми, не може перевищувати 128 символів, що є DOS-обмеженням.

1.17 Складені оголошення

Прості оголошення в мові Сі дозволяють визначати прості змінні, масиви, покажчики, функції, структури та об'єднання.

В найпростішому випадку, якщо оголошується проста змінна базового типу, типу структури або об'єднання, ідентифікатор описується типу, що заданий специфікацією типу.

Для оголошення масиву значень деякого типу, функції, що повертає значення деякого типу, або покажчика на значення деякого типу, ідентифікатор доповнюється відповідно квадратними дужками […] справа, круглими дужками (…) справа або ознакою покажчика – зірочкою (*) зліва.

Наступні приклади ілюструють найпростіші форми оголошень :

```
int list[20];  /* масив list із 20 цілих значень */
char *cp;      /* покажчик cp на значення типу char */
double func(); /* функція func(), що повертає значення
                  типу double*/
```

Синтаксис оголошення :

[клас_пам'яті] тип ідентифікатор [= ініціалізатор] ;

Оголошення складаються з чотирьох частин :

1. необов'язкового специфікатора класу пам'яті (*auto*, *register*, *static*, *extern*);
2. базового типу або типу користувача;
3. оголошуючої частини;
4. необов'язкового ініціалізатора.

Оголошуюча частина в свою чергу складається з ідентифікатора і, можливо, *операторів оголошення*. Найчастіше використовуються наступні оператори оголошення (таблиця 1.18).

Таблиця 1.18. „Оператори оголошення"

*	покажчик	префікс
*const	константний покажчик	префікс
&	посилання (адреса)	префікс
[ ]	масив	суфікс
( )	функція	суфікс

Суфіксні оператори оголошення „міцніше зв'язані" з ім'ям, ніж префіксні. Тому *typename *str[];* означає масив покажчиків на деякі об'єкти, а для визначення типів таких як „покажчик на функцію" необхідно використовувати дужки.

Складене оголошення – це ідентифікатор, що доповнений більше ніж однією ознакою масиву, покажчика, або функції.

З одним ідентифікатором можна створити множину різних комбінацій ознак типу масив, покажчик або функція. Причому, деякі комбінації неприпустимі. Наприклад, масив не може містити в якості елементів функцію, а функція не може повертати масив або функцію.

При інтерпретації складених оголошень спочатку розглядають квадратні і круглі дужки, що розташовані справа від ідентифікатора. Квадратні і круглі дужки мають однаковий пріоритет. Вони інтерпретуються зліва направо. Після них розглядаються зірочки, що розташовані зліва від ідентифікатора. Специфікація типу розглядається на останньому етапі, після того, як все складене оголошення проінтерпретоване.

Круглі дужки можуть також використовуватися для зміни існуючого по замовчуванню порядку інтерпретації оголошення. Наприклад :

```
int *func();  /*функція, що повертає покажчик на int */
int (*func)();/*покажчик на функцію, що повертає int */
```

Алгоритм інтерпретації складених оголошень :

1. Знайти ідентифікатор (якщо їх декілька, то необхідно почати з того, який знаходить ближче до „середини" складеного оголошення).
2. Подивитися вправо :
   - Якщо справа розташована відкриваюча кругла дужка – тоді це функція, а вираз, що розташований між цією відкриваючою дужкою '(' і відповідною їй закриваючою дужкою ')' необхідно інтерпретувати як параметри функції.
   - Якщо справа стоїть відкриваюча квадратна дужка '[' – тоді це масив і вираз між відповідними квадратними дужками […] необхідно інтерпретувати як розмір масиву. *Примітка* : якщо масив багатовимірний, то за дужками […] розташовується ще одна або декілька серій квадратних дужок […].
   - Якщо на будь-якому етапі інтерпретації справа зустрічається закриваюча кругла дужка ')', то необхідно спочатку повністю

провести інтерпретацію всередині даної пари круглих дужок, а потім подовжити інтерпретацію справа від закриваючої круглої дужки ‘)’.

3. Якщо зліва від проінтерпретованого виразу розташована зірочка і :
   - проінтерпретований вираз є функцією, то вона повертає покажчик;
   - проінтерпретований вираз є масивом, то кожний елемент цього масиву є покажчиком;
   - проінтерпретований вираз не є ні функцію, ні масивом, то вираз є покажчиком.
4. Застосувати описані вище правила (2-3 пункт алгоритму) ще раз.
5. Проінтерпретувати специфікацію типу даних.

Приклад інтерпретації складених оголошень :

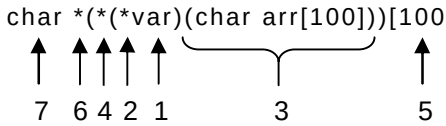


Рис. 1.22. Порядок інтерпретації складного оголошення

1. Ідентифікатор *var* оголошений як
2. покажчик на
3. функцію, що приймає в якості аргументу масив із ста значень типу char і повертає
4. покажчик на
5. масив із ста елементів, кожний з яких є
6. покажчиком на
7. значення типу *char.*

1.17.1 Описи з модифікаторами

Використання в оголошеннях спеціальних ключових слів (модифікаторів) дозволяють надавати оголошенням спеціального змісту. Інформація, яку несуть в собі модифікатори, використовуються компілятором мови Сі в процесі генерування коду.

Розглянемо правила інтерпретації оголошень, що містять модифікатори *const*, *volatile*, *cdecl*, *pascal*, *near*, *far*, *huge*, *interrupt*.

Модифікатори *cdecl*, *pascal*, *interrupt* повинні розташовуватися безпосередньо перед ідентифікатором.

Модифікатори *const*, *volatile*, *near*, *far*, *huge* впливають або на ідентифікатор, або на ознаку покажчика (зірочку), що розташована безпосередньо справа від модифікатора. Якщо справа розташований ідентифікатор, то модифікується тип об'єкта, що іменується даним ідентифікатором. Якщо ж справа розташована зірочка, то ця зірочка представляє собою покажчик на модифікований тип. Таким чином, конструкція

модифікатор *

читається як „покажчик на модифікований тип".

Наприклад,

```
int const *p; /* покажчик на цілу константу */
int *const p; /* константний покажчик на величину типу
int */
```

Модифікатори типу *const* і *volatile* можуть також розташовуватися і перед специфікацією типу.

В TC використання модифікаторів *near*, *far*, *huge* обмежене: вони можуть бути записані тільки перед ідентифікатором функції або перед ознакою покажчика (зірочкою).

Допускається більше одного модифікатора для одного об'єкта (або елемента оголошення). В наступному прикладі тип функції *func* модифікується одночасно спеціальними ключовими словами *far* і *pascal*. Порядок ключових слів неважливий, тобто комбінації *far pascal* і *pascal far* мають однаковий зміст.

```
int far * pascal far func();
```

Тип значення, що повертається функцією *func*, представляє собою покажчик на значення типу *int*. Тип цих значень модифікований спеціальним ключовим словом *far*.

Як і звичайно, в оголошенні можуть бути використані круглі дужки для зміни порядку його інтерпретації.

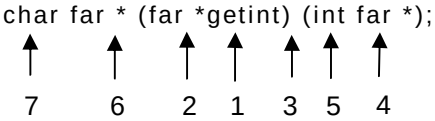


Рис. 1.23. Порядок інтерпретації складного оголошення з модифікаторами

В даному прикладі наведене оголошення з різними варіантами розташування модифікатора *far*. Враховуючи правило, відповідно до якого модифікатор впливає на елемент оголошення, розташований справа від нього, можна інтерпретувати це оголошення наступним чином.

1. Ідентифікатор *getint* оголошений як
2. покажчик на *far*
3. функцію, що приймає
4. один аргумент, який є покажчиком на *far*
5. значення типу *int*
6. і повертає покажчик на *far*
7. значення типу *char*

1.17.2 Модифікатори const і volatile

Про модифікатор *const* йшла мова в розділі 1.2.3.”Константи”. Модифікатор *const* не допускає явного присвоювання змінній або інших дій, що можуть вплинути на зміну її значення, таких як виконання операції інкременту і декременту. Значення покажчика, що оголошений з модифікатором *const*, не може бути зміненим, на відміну від значення об’єкта, на який він вказує.

Модифікатори *volatile* і *const* протилежні за змістом.

Модифікатор *volatile* вказує на те, що значення змінної може бути зміненим; але не тільки безпосередньо програмою, а також і зовнішнім впливом (наприклад, програмою обробки переривань, або, якщо змінна відповідає порту введення/виведення, обміном із зовнішнім пристроєм). Оголошення об’єкта з модифікатором *volatile* попереджує компілятор мови Сі, чого не слід робити

Можливим також є одночасне використання в оголошенні модифікаторів *const* і *volatile*. Це означає, що значення змінної не може модифікуватися програмою, але піддається зовнішньому впливу.

Якщо з модифікатором *const* або *volatile* оголошується змінна складеного типу, то дія модифікатора розповсюджується на всі його складові елементи.

Примітка. При відсутності в оголошенні специфікації типу і присутності модифікатора *const* або *volatile* мається на увазі тип *int*.

Приклади:

```
float const pi=3.14159265;
const maxint=32767;
char *const str= "Деякий рядок."; /* покажчик-константа */
char const *str2= „Рядок";/* покажчик на константний рядок */
```

Із врахуванням наведених вище оголошень наступні оператори неприпустимими.

```
pi=3.0;          /* присвоювання значення константі */
i=maxint--;      /* зменшення константи */
str="Other string"; /* присвоювання значення константі-
                       покажчику */
```

Однак виклик функції strcpy(str,"String"); припустимий, так як в даному випадку здійснюється посимвольне копіювання рядка в область пам'яті, на яку вказує покажчик.

Аналогічно, якщо покажчик на тип *const* присвоїти покажчику на тип, відмінний від *const*, то через отриманий покажчик можна здійснювати присвоювання.

1.17.3 Модифікатори cdecl і pascal

Результатом роботи компілятора мови Сі є файл, що містить об'єктний код програми. Файли з об'єктним кодом, що отримуються в результаті компіляції всіх файлів програми, компоновщик об'єднує в один файл виконання.

При компіляції всі глобальні ідентифікатори програми, тобто імена функцій і глобальних змінних, зберігаються в об'єктному коді і використовуються компоновщиком в процесі роботи. По замовчуванню ці ідентифікатори зберігаються в своєму початковому вигляді. Крім того, в якості першого символу кожного ідентифікатора компілятор мови Сі додає символ підкреслення.

Компоновщик по замовчуванню розрізняє великі та малі літери, тому ідентифікатори, що використовуються в різних файлах програми для іменування одного і того самого об'єкта, повинні повністю співпадати з точки зору як орфографії, так і регістрів літер. Для здійснення співпадіння ідентифікаторів, що використовуються в різномовних файлах, використовуються модифікатори *pascal* і *cdecl*.

Використання модифікатора *pascal* до ідентифікатора призводить до того, що ідентифікатор перетворюється до верхнього регістру і до нього не додається символ підкреслення. Цей ідентифікатор не може

використовуватися для іменування в програмі на мові Сі глобального об'єкта, який використовується також в програмі на мові Паскаль. В об'єктному коді, що згенерований компілятором мови Сі, і в об'єктному коді, що згенерований компілятором мови Паскаль, ідентифікатор буде представлений ідентично.

Якщо модифікатор *pascal* застосовується до ідентифікатора функції, то він здійснює вплив також і на передачу аргументів функції. Засилання аргументів у стек здійснюється в цьому випадку не в оберненому порядку, як прийнято в компіляторах мови Сі, а в прямому – першим засилається в стек перший аргумент.

Функція типу *pascal* не може мати змінне число параметрів, як, наприклад, функція *printf*().

Існує ще один модифікатор, яка присвоює всім функціям і покажчикам на функції тип *pascal*. Це означає, що вони будуть використовувати послідовність виклику, що прийнята в мові Паскаль, а їх ідентифікатори будуть можливими для виклику з програми на Паскалі. При цьому можна сказати, що деякі функції і покажчики на функції використовують викликаючу послідовність, прийняту в мові Сі, а їх ідентифікатори мають традиційний вигляд для ідентифікаторів мови Сі. Для цього їх оголошення повинні містити модифікатор *cdecl*.

1.17.4 Модифікатори near, far, huge

Ці модифікатори здійснюють вплив на роботу з адресами об'єктів.

Компілятор мови Сі дозволяє використовувати при компіляції одну з декількох моделей пам'яті.

Використання моделі пам'яті визначає розміщення програми і даних в ОП, а також внутрішній формат покажчиків. Однак при використанні будь-якої моделі пам'яті можна оголосити покажчик з форматом, що відрізняється від прийнятого по замовчуванню. Це здійснюється за допомогою модифікаторів *near, far* і *huge*.

Покажчик типу *near* – 16-бітовий; для визначення адреси об'єкта він використовує зсув відносно поточного вмісту сегментного регістру. Для покажчика типу near доступна пам'ять обмежена розміром поточного 64-кілобайтного сегмента даних.

Покажчик типу *far* – 32-бітовий; він містить як адресу сегменту, так і зсув. При використанні покажчиків типу *far* припустимі звернення до пам'яті в межах 1-мегабайтного адресного простору, однак значення покажчика типу far циклічно змінюється в межах 64-кілобайтного сегменту.

Покажчик типу *huge* – 32-бітовий; він також містить адресу сегменту і зсув. Значення покажчика типу *huge* може бути змінене в межах 1-мегабайтного адресного простору. В ТС покажчик *huge* завжди зберігається в нормалізованому форматі.

1.17.5 Модифікатор interrupt

Модифікатор *interrupt* призначений для оголошення функцій, що працюють з векторами переривань процесору. Для функції типу *interrupt* при компіляції генерується додатковий код в точці входу і виходу з функції, для збереження і відновлення значень регістрів процесора.

Функції переривань слід оголошувати з типом повернення *void*. Модифікатор *interrupt* не може використовуватися спільно з модифікаторами *near*, *far* та *huge*.

1.18 Директиви препроцесора

Обробка програми препроцесором здійснюється перед її компіляцією. На цьому етапі попередньої обробки можуть виконуватися наступні дії : включення у файл, що компілюється інших файлів, визначення символічних констант і макросів, встановлення режиму умовної компіляції програми і умовного виконання директив препроцесора. Директивами називаються інструкції препроцесора. Всі директиви повинні починатися з символу #, перед яким в рядку можуть розташовуватися тільки пробільні символи.

Примітка. Після директив препроцесора крапка з комою не ставиться.

1.18.1 Директива #include

Синтаксис :

#include “ім’я_файла”

#include <ім’я_файла>

Директива #include використовується для включення копії вказаного файла в те місце програми, де знаходиться ця директива.

Різниця між двома формами директиви полягає в методі пошуку пре процесором файла, що включається. Якщо ім’я файла розміщене в „кутових” дужках < >, то послідовність пошуку препроцесором заданого файла в каталогах визначається встановленими каталогами

включення (include directories). Якщо ж ім'я файла заключне в лапки, то препроцесор шукає в першу чергу файл у поточній директорії, а потім вже у каталогах включення.

Робота директиви #include зводиться практично до того, що директива #include прибирається, а на її місце заноситься копія вказаного файла.

Текст файла, що включається може містити директиви препроцессора, і директиву #include зокрема. Це означає, що директива #include може бути вкладеною. Допустимий рівень вкладеності директиви #include залежить від конкретної реалізації компілятора.

```
#include <stdio.h> /* приклад 1*/
#include "defs.h"  /* приклад 2*/
```

В першому прикладі у головний файл включається файл з ім'ям *stdio.h*. Кутові дужки повідомляють компілятору, що пошук файла необхідно здійснювати в директоріях, вказаних в командному рядку компіляції, а потім в стандартних директоріях.

В другому прикладі в головний файл включається файл з ім'ям *defs.h*. Подвійні лапки означають, що при пошуку файла спочатку повинна бути переглянута директорія, що містить поточний файл.

В TC є також можливість задавати ім'я шляху в директиві #include за допомогою іменованої константи. Якщо за словом *include* слідує ідентифікатор, то препроцесор перевіряє, чи не іменує він константу або макровизначення. Якщо ж за словом *include* слідує рядок, що заключений в лапки або в кутові дужки, то TC не буде шукати в ній ім'я константи.

```
#define myincl "c:\test\my.h"

#include myincl
```

1.18.2 Директива #define

Синтаксис :

#define ідентифікатор текст

#define ідентифікатор (список_параметрів) текст

Директива #define заміняє всі входження *ідентифікатора* у програмі на *текст*, що слідує в директиві за *ідентифікатором*. Цей процес називається макропідстановкою. *Ідентифікатор* замінюється лише в тому випадку, якщо він представляє собою окрему лексему. Наприклад, якщо *ідентифікатор* є частиною рядка або більш довгого

ідентифікатора, він не замінюється. Якщо за *ідентифікатором* слідує список параметрів, то директива визначає макровизначення з параметрами.

Текст представляє собою набір лексем, таких як ключові слова, константи, ідентифікатори або вирази. Один або більше пробільних символів повинні відділяти *текст* від *ідентифікатора* (або заключених в дужки параметрів). Якщо *текст* не вміщується в рядку, то він може бути продовжений на наступному рядку; для цього слід набрати в кінці рядка символ обернений слеш \ і зразу за ним натиснути клавішу *Enter*.

Текст може бути опущений. В такому разі всі екземпляри *ідентифікатора* будуть вилучені з тексту програми. Але сам ідентифікатор розглядається як визначений і при перевірці директива #if дає значення 1.

Список параметрів, якщо він заданий, містить один або більше *ідентифікаторів*, розділених комами. *Ідентифікатори* в рядку параметрів повинні відрізнятися один від одного. Їх область дії обмежена макровизначенням. Список параметрів повинен бути заключений в круглі дужки. Імена формальних параметрів у *тексті* відмічають позиції, в які повинні бути підставлені фактичні аргументи макровиклику. Кожне ім'я формального параметра може з'явитися в тексті довільне число разів.

В макровиклику вслід за *ідентифікатором* записується в круглих дужках список фактичних аргументів, що відповідають формальних параметрам із *списку параметрів*. *Текст* модифікується шляхом заміни кожного формального параметра на відповідний фактичний параметр. Списки фактичних параметрів і формальних параметрів повинні мастити одне і те ж число елементів.

Примітка. Не слід плутати підстановку аргументів в макровизначеннях з передачею параметрів у функціях. Підстановка в препроцесорі носить чисто текстовий характер. Ніяких обчислень при перетворенні типу при цьому не виконується.

Вище вже говорилося, що макровизначення може містити більше одного входження даного формального параметра. Якщо формальний параметр представлений виразом з "побічним ефектом" і цей вираз буде обчислюватися більше одного разу, разом з ним кожний раз буде виникати і "побічний ефект". Результат виконання в цьому випадку може бути помилковим.

Всередині *тексту* в директиві #define можуть знаходитися вкладені імена інших макровизначень або констант.

Після того, як виконана макропідстановка, отриманий рядок знову переглядається для пошуку інших імен констант і макровизначень. При повторному перегляді не розглядається ім'я раніше проведеної макропідстановки. Тому директива

```
#define a a
```

не призведе до за циклювання препроцесора.

Приклад 1 :

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
```

В даному прикладі ідентифікатор WIDTH визначається як ціла константа із значенням 80, а ідентифікатор LENGTH – як текст (WIDTH+10). Кожне входження ідентифікатора LENGTH у програму буде замінено на текст (WIDTH+10), який після розширення ідентифікатора WIDTH перетвориться на вираз (80+10). Дужки дозволяють уникнути помилок в операторах, подібних наступному :

```
val=LENGTH*20;
```

Після обробки програми препроцесором текст набуде вигляду :

```
val=(80+10)*20;
```

Значення, яке буде присвоєно змінній *val* рівне 1800. При відсутності дужок значення *val* буде рівне 280.

```
val=80+10*20;
```

Приклад 2 :

```
#define MAX(x,y) ((x)>(y))?(x):(y)
```

В даному прикладі визначається макровизначення MAX. Кожне входження ідентифікатора MAX в тексті програми буде замінено на вираз ((x)>(y))?(x):(y), в якому замість формальних параметрів *x* та *y* підставляються фактичні. Наприклад, макровиклик :

```
MAX(1,2)
```

заміниться на вираз ((1)>(2))?(1):(2).

1.18.3 Директива #undef

Синтаксис :

#undef ідентифікатор

Визначення символічних констант і макросів можуть бути анульовані за допомогою директиви препроцесора #undef. Таким чином, область дії символічної константи або макросу починається з місця їх визначення і закінчується явним їх анулюванням директивою #undef або кінцем файла.

Після анулювання ідентифікатор може бути знову використаний директивою #define.

Приклад :

```
#define WIDTH 80
/* … */
#undef WIDTH
/* … */
#define WIDTH 20
```

1.18.4 Директиви #if, #elif, #else, #endif

Умовна компіляція дає можливість програмісту керувати виконанням директив препроцесора і компіляцією програмного коду. Кожна умовна директива препроцесора обчислює значення цілочисельного константного виразу.

Умовна директива препроцесора #if багато в чому схожа на оператор *if*. Її синтаксис має вигляд :

```
#if  умова
...
[ #elif  умова
…]
[ #elif  умова
…]
[ #else
…]
#endif
```

Умова – це цілочисельний вираз. Якщо цей вираз повертає не нуль (істинно), то фрагмент коду, що розташований між директивою #if і директивою #endif, компілюється. Якщо ж вираз повертає нуль (хибно), то цей фрагмент коду ігнорується і препроцесором, і компілятором.

В умовах, окрім звичайних виразів, можна використовувати конструкцію :

```
defined (ідентифікатор)
```

defined повертає 1, якщо вказаний ідентифікатор раніше був визначений директивою #define, і повертає 0 в протилежному випадку.

Кількість директив #elif – довільна. Якщо директива #else присутня, то між нею і директивою #endif на даному рівні вкладеності не повинно бути інших директив #elif.

Приклад 1:

```
#if defined(CREDIT)
      credit();
#elif defined (DEBIT)
      debit();
#else
      printerror();
#endif
```

В наведеному прикладі директиви #if, #elif, #else, #endif керують викликом однієї з трьох викликів функцій. Виклик функції *credit*() скомпілюється, якщо визначена іменована константа CREDIT. Якщо визначена іменована константа DEBIT, то скомпілюється виклик функції *debit*(). Якщо жодна із наведених іменованих констант не визначена, то скомпілюється виклик функції *printerror*().

Приклад 2.

```
#if DLEVEL>5
#define SIGNAL 1
#if STACKUSE == 1
#define STACK 200
#else
#define STACK 100
#endif
#else
#define SIGNAL 0
#if STACKUSE == 1
#define STACK 100
#else
#define STACK 50
#endif
#endif
```

В другому прикладі показано два вкладених набори директив #if, #else, #endif. Перший набір директив оброблюється, якщо значення DLEVEL більше за 5. В протилежному випадку оброблюється другий набір.

1.18.5 Директиви #ifdef і #ifndef

Синтаксис :

#ifdef ідентифікатор
#ifndef ідентифікатор

Аналогічно директиві #if, за директивами #ifdef і #ifndef може слідувати набір директив #elif і директива #else. Набір директив повинен закінчуватися директивою #endif.

Використання директив #ifdef і #ifndef еквівалентне директиві #if, що використовує вираз з операцією defined (ідентифікатор). Ці директиви підтримуються виключно для сумісності з попередніми версіями компіляторів мови Сі. Тому замість цих директив рекомендується використовувати директиву #if з операцією defined (ідентифікатор).

Коли препроцесор оброблює директиву #ifdef, він перевіряє, чи визначений в даний момент вказаний ідентифікатор. Якщо так, то умова вважається істинною, якщо ні – хибною.

Директива #ifndef протилежна за своєю дією директиві #ifdef. Якщо ідентифікатор не був визначений директивою #define, або його дія відмінена директивою #undef, то умова вважається істинною. В протилежному випадку умова хибна.

1.18.6 Директива #line

Синтаксис :

#line константа [“ім’я_файла”]

Директива препроцесора #line повідомляє компілятору мови Сі про зміну імені програми і порядку нумерації рядків. Ці зміни відбиваються лише на повідомленнях компілятора : файл програми буде тепер іменуватися як „ім’я_файла”, а поточний рядок, що компілюється отримує номер, вказаний константою. Після обробки чергового рядка лічильник номера рядка збільшується на одиницю. У випадку зміни номера рядка й імені файла програми директивою #line, компілятор „забуває” їх попереднє значення і продовжує роботу вже з новими значеннями.

Поточний номер рядка і ім’я файла програми доступні в програмі через псевдозмінні з іменами __LINE__ і __FILE__. Ці псевдозмінні

можуть використовуватися для виведення під час виконання програми повідомлень про точне місцезнаходження помилки.

Значенням псевдозмінної __FILE__ є рядок, що представляє ім'я файла, заключене в подвійні лапки. Тому для друку імені файлу програми не потрібно заключати сам ідентифікатор __FILE__ в подвійні лапки.

1.19 Динамічні структури даних

Незважаючи на те, що терміни *тип даних* та *структура даних* звучать дещо схоже, проте вони мають різний підтекст.

Як говорилося раніше, *тип даних* – це множина значень, які може приймати змінна деякого типу. А с*труктури даних* представляють собою набір даних, можливо різних типів, що об'єднані певним чином.

Базовим елементом структури даних є елемент (вузол), який призначений для зберігання певного типу даних. Якщо елементи зв'язані між собою за допомогою покажчиків, то такий спосіб організації даних називається *динамічними структурами даних,* так як їх розмір динамічно змінюється під час виконання програми.

З динамічних структур даних найчастіше використовуються лінійні списки, стеки, черги та бінарні дерева.

1.19.1 Лінійні списки

Лінійний список – це скінченна послідовність однотипних елементів (вузлів). Кількість елементів у цій послідовності називається довжиною списку. Наприклад :

F=(1,2,3,4,5,6) – лінійний список, його довжина 6.

При роботі зі списками дуже часто доводиться виконувати такі операції :

- додавання елемента в початок списку;
- вилучення елемента з початку списку;
- додавання елемента в будь-яке місце списку;
- вилучення елемента з будь-якого місця списку;
- перевірку, чи порожній список;
- очистку списку;
- друк списку.

Основні методи зберігання лінійних списків поділяються на *методи послідовного* та *зв'язаного зберігання*.

Послідовне зберігання списків. Метод послідовного зберігання списків ґрунтується на використанні масиву елементів деякого типу та змінної, в якій зберігається поточна кількість елементів списку.

```
#define MAX 100    /* максимально можлива довжина
списку */
typedef struct
{
  int x; /* тут потрібно описати структуру елементів
списку*/
} elementtype;
typedef struct
{
  elementtype elements[MAX];
  int count;
} listtype;
```

У вищенаведеному фрагменті програми описуються типи даних *elementtype* (визначає структуру елемента списку) та *listtype* (містить масив для зберігання елементів та змінну для зберігання поточного розміру списку).

Наводимо приклади реалізації функцій для виконання основних операцій над списками.

1. *Ініціалізація списку* (робить список порожнім).

```
void list_reset(listtype *list)
{
  list->count=0;
};
```

2. Додавання нового елементу у кінець списку.

```
void list_add(listtype *list,elementtype element)
{
  if (list->count==MAX) return;
  list->elements[list->count++]=element;
};
```

3. Додавання нового елементу в позицію pos.

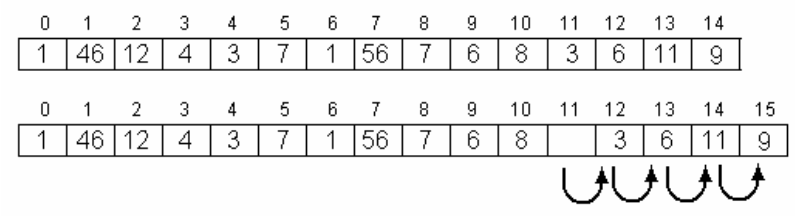


Рис. 1.24. Додавання нового елемента в задану позицію

```
void list_insert(listtype *list,int pos,elementtype
element)
{
  int j;
  if (pos<0||pos>list->count||pos>=MAX) return;
  for (j=list->count;j>pos;j--)
  {
    list->elements[j]=list->elements[j-1];
  };
  list->elements[j]=element;
  list->count++;
};
```

4. Вилучення елемента з номером pos.

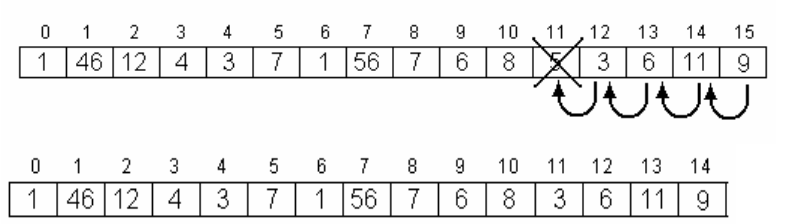


Рис.1.25. Схема вилучення елемента зі списку

```
void list_delete(listtype *list,int pos)
{
  int j;
  if (pos<0||pos>list->count) return;
  for (j=pos+1;j<list->count;j++)
  {
    list->elements[j-1]=list->elements[j];
  };
  list->count--;
};
```

5. Отримання елемента з номером pos.

```
int list_get(listtype *list,int pos,elementtype
*element)
{
  if (pos<0||pos>list->count)
  {
    return 0;
  };
  *element=list->elements[pos];
  return 1;
};
```

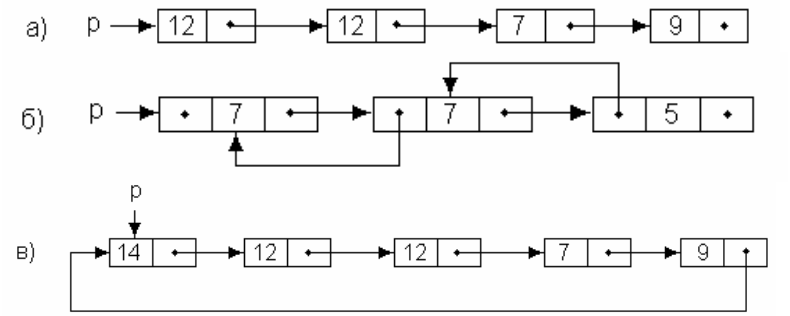


Рис. 1.26. Схематична структура односпрямованого (а), двоспрямованого (б), та кільцевого списків (в)

При послідовному зберіганні списків за допомогою масивів елементи списку зберігаються в масиві. Така організація дозволяє легко переглядати вміст списку та додавати нові елементи в його кінець. Але такі операції, як вставка нового елемента в середину списку чи вилучення елементу з середини списку потребують зсуву всіх наступних елементів. При збільшенні елементів масиву кількість операцій, потрібна для впорядкування списку, стрімко зростає.

Зв'язане зберігання лінійних списків. Найпростіший спосіб зв'язати множину елементів – зробити так, щоб кожний елемент

містив посилання на наступний. Такий список називається *односпрямованим (однозв'язаним).* Якщо додати в такий список ще й посилання на попередній елемент, то отримаємо двозв'язаний список. А список, перший та останній елементи якого зв'язані, називається кільцевим.

Структуру даних для зберігання односпрямованого лінійного списку можна описати таким чином :

```
typedef long elemtype;
typedef struct node
{
  elemtype val;
  struct node *next;
} list;
```

В даному фрагменті програми описуються декілька типів даних :

elemtype – визначає тип даних лінійного списку. Можна використовувати будь-який стандартний тип даних, включаючи структури.

list – визначає структуру елемента лінійного списку (*val* – значення, яке зберігається у вузлі, *next* – покажчик на наступний вузол).

Схематично лінійний односпрямований список виглядає так :

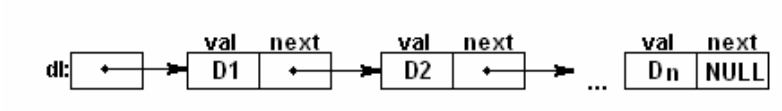


Рис. 1.27. Схематичне зображення односпрямованого лінійного списку

Реалізація основних операцій :

1. Включення елемента в початок списку.

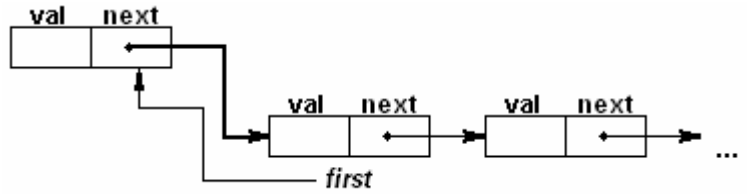


Рис. 1.28. Схема дії операції включення елемента в початок списку

```
list *addbeg(list *first, elemtype x){
      list *vsp;
      vsp = (list *) malloc(sizeof(list));
      vsp->val=x;
      vsp->next=first;
      first=vsp;
   return first;
}
```

2. Видалення елемента з початку списку.

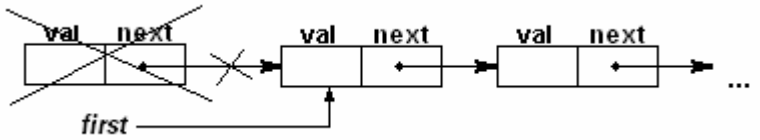


Рис. 1.29. Схема дії операції видалення елемента

```
list *delbeg(list *first)
{  list *vsp;
   vsp=first->next;
   free(first);
   return vsp;
}
```

3. Включення нового елемента у список.

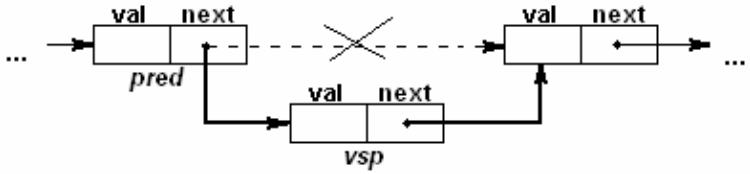


Рис. 1.30. Схема включення нового елемента у список

```
list *add(list *pred, elemtype x)
{  list *vsp;
   vsp = (list *) malloc(sizeof(list));
   vsp->val=x;
   vsp->next=pred->next;
   pred->next=vsp;
   return vsp;
}
```

4. Видалення елемента зі списку.

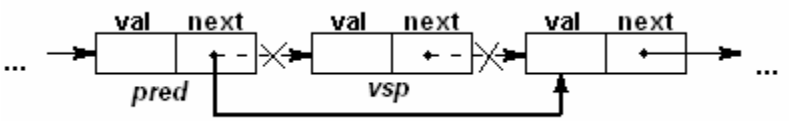


Рис. 1.31. Схема вилучення елемента зі списку

```
elemtype del(list *pred)
{  elemtype x;
   list *vsp;
   vsp=pred->next;
   pred->next=pred->next->next;
   x=vsp->val;
   free(vsp);
   return x;
}
```

5. Друк значень списку.

```
void print(list *first)
{ list *vsp;
  vsp=first;
  while (vsp)
  {
     printf("%i\n",vsp->val);
     vsp=vsp->next;
  }
}
```

6. Перевірка, чи порожній список

```
int pust(list *first)
{
   return !first;
}
```

7. Знищення списку

```
list *kill(list *first)
{
   while (!pust(first)) first=delbeg(first);
   return first;
}
```

1.19.2 Стеки

Стек — динамічна структура даних, яка представляє собою впорядкований набір елементів, в якому додавання нових елементів і видалення існуючих проходить з одного кінця, який називається вершиною стека.

Стек реалізує принцип LIFO (last in – first out, останнім прийшов – першим пішов). Найбільш наглядним прикладом організації стеку може бути дитяча пірамідка, де додавання і знімання кілець здійснюється як раз відповідно до цього принципу.

Основні операції, які можна виконувати над стеками :

- додавання елемента в стек;
- вилучення елемента із стека;
- перевірка, чи порожній стек;
- перегляд елемента у вершині стека без видалення;
- очистка стека.

Стек створюється так само, як і лінійний список, так як стек є частковим випадком односпрямованого списку.

```
typedef long elemtype;
typedef struct node
{
   elemtype val;
   struct node *next;
} stack;
```

Реалізація основних операцій над стеками :

1. Початкове формування стеку

```
stack *first(elemtype d)
{
  stack *pv=(stack*) calloc(1,sizeof(stack));
  pv->val=d;
  pv->next=NULL;
  return pv;
};
```

2. Занесення значення в стек

```
void push(stack **top,elemtype d)
{
  stack *pv=(stack*) calloc(1,sizeof(stack));
  pv->val=d;
  pv->next=*top;
  *top=pv;
};
```

3. Вилучення елемента зі стека

```
elemtype pop(stack **top)
{
   elemtype temp=(*top)->val;
   stack *pv=*top;
   *top=(*top)->next;
   free(pv);
   return temp;
};
```

1.19.3 Черги

Черга - це лінійний список, де елементи вилучаються з початку списку, а додаються в кінець (як звичайна черга в магазині).

Двостороння черга - це лінійний список, у якого операції додавання, вилучення і доступу до елементів можливі як спочатку так і в кінці списку. Таку чергу можна уявити як послідовність книг, що стоять на полиці так, що доступ до них можливий з обох кінців.

Черга є частковим випадком односпрямованого списку. Вона реалізує принцип FIFO (first in – first out, першим прийшов – першим пішов).

Черги створюються аналогічно до лінійних списків та стеків.

```
typedef long elemtype;

typedef struct node
{
  elemtype val;
  struct node *next;
} queue;
```

1. Початкове формування черги

```
queue *first (elemtype d)
{
  queue *pv=(queue*) calloc(1,sizeof(queue));
  pv->val=d;
  pv->next=NULL;
  return pv;
}
```

2. Додавання елемента в кінець

```
void add (queue **pend, elemtype d)
{
  queue *pv=(queue*) calloc(1,sizeof(queue));
  pv->val=d;
  pv->next=NULL;
  (*pend)->next=pv;
  *pend=pv;
}
```

3. Вилучення елемента з кінця

```
elemtype del(queue **pbeg)
{
  elemtype temp=(*pbeg)->val
  queue *pv=*pbeg;
  *pbeg=(*pbeg)->next;
  free(pv)
  return temp;
}
```

1.19.4 Двійкові дерева

Бінарне дерево – це динамічна структура даних, що складається з вузлів (елементів), кожен з яких містить, окрім даних, не більше двох посилань на інші бінарні дерева. На кожен вузол припадає рівно одне посилання. Початковий вузол називається коренем дерева (рис 1.23.).

Якщо дерево організоване таким чином, що для кожного вузла всі ключі його лівого піддерева менші за ключ цього вузла, а всі ключі його правого піддерева – більші, воно називається *деревом пошуку*. Однакові ключі в деревах пошуку не допускаються.

В дереві пошуку можна знайти елемент за ключем, рухаючись від кореня і переходячи на ліве або праве піддерево в залежності від значення ключа в кожному вузлі. Такий спосіб набагато ефективніший пошуку по списку, так як час виконання операції пошуку визначається висотою дерева.

Дерево є рекурсивною структурою даних, так як кожне піддерево є також деревом. Дії з такими структурами даних простіше всього описувати за допомогою рекурсивних алгоритмів.

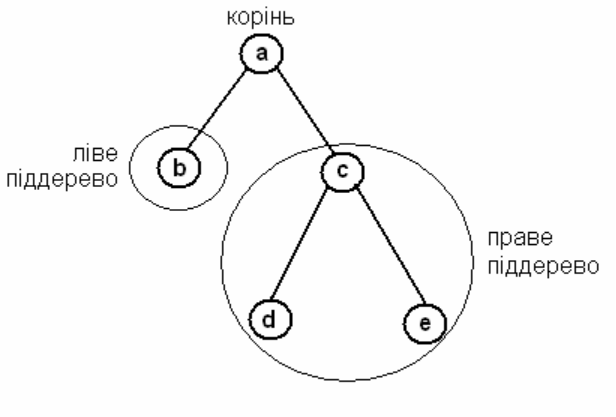


Рис. 1.32. Схематичне зображення дерева

Наприклад, функцію обходу всіх вузлів дерева в загальному вигляді можна описати так :

```
function way(дерево)
{
  way(ліве піддерево);
  обробка кореня;
  way(праве піддерево);
};
```

Можна обходити дерево і в іншому порядку, наприклад, спочатку корінь, а потім піддерева. Але наведена модель функції дозволяє отримати на виході відсортовану послідовність ключів, так як спочатку відвідуються вершини з меншими ключами, що розташовані в лівому піддереві.

Таким чином, дерева пошуку можна використовувати для сортування значень. При обході дерева вузли не видаляються.

Для бінарних дерев визначені наступні операції :

- включення вузла у дерево;
- пошук по дереву;
- обхід дерева;
- видалення вузла.

Для кожного рекурсивного алгоритму можна створити його нерекурсивний еквівалент.

Вузол бінарного дерева можна визначити як :

```
typedef struct sbtree
{
  int val;
  struct sbtree *left,*right;
} btree;
```

Реалізація деяких операцій з бінарними деревами.

1). Рекурсивний пошук в бінарному дереві. Функція повертає покажчик на знайдений вузол.*:*

```
btree *Search(btree *p, int v)
{
  if (p==NULL) return(NULL); /* вітка порожня */
  if (p->val == v) return(p);  /* вершина знайдена */
  if (p->val > v)  /* порівняння з поточним   вузлом */
    return(Search(p->left,v));  /* ліве піддерево */
  else
    return(Search(p->right,v));  /* праве піддерево */
}
```

2). Включення значення в двійкове дерево (рис 1.33.)*:*

```
btree *Insert(btree *pp, int v)
{
  if (pp == NULL) /* знайдена порожня вітка */
  {
     btree *q = (btree*) calloc(1,sizeof(btree));
              /* створити вершину дерева  */
     q->val = v; /* і повернути покажчик */
     q->left = q->right = NULL;
     return q;
  }
  if (pp->val == v) return pp;
  if (pp->val > v)  /* перейти в ліве піддерево */
    pp->left=Insert(pp->left,v);
  else
    pp->right=Insert(pp->right,v);
                        /* перейти в праве піддерево*/
  return pp;
}
```

3). Рекурсивний обхід двійкового дерева :

```
void Scan(btree *p)
{
  if (p==NULL) return;
  Scan(p->left);
  printf("%i\n",p->val);
  Scan(p->right);
}
```

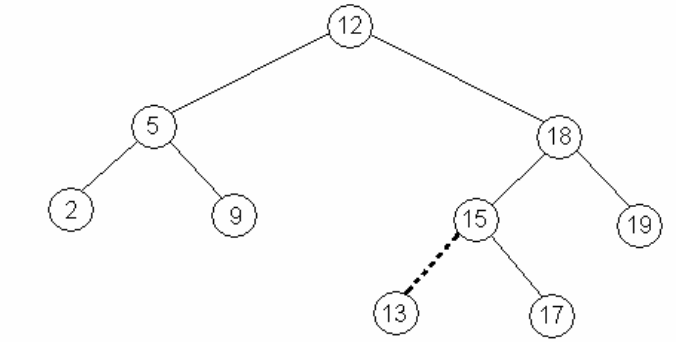


Рис. 1.33. Схема включення нового елемента у бінарне дерево

ЧАСТИНА 2. МОВА ПРОГРАМУВАННЯ Сі++

2.1 Історія виникнення

Мова програмування *Сі++* була створена Б'єрном Страуструпом на основі мови *Сі* – одного з найпоширеніших інструментальних засобів програмування. Вона містить основні типи даних, операції, синтаксис та структуру програми мови Сі, додаючи суттєво нове – підтримку *абстрактних типів даних (**АТД**)* та *об'єктно-орієнтованого програмування (**ООП**)*.

Всі ми живемо у світі об'єктів, що існують довкола, проте кожний з них має свої властивості та ознаки, а також множину своїх допустимих операцій. Об'єкт у програмуванні, як і в повсякденному житті, являє собою сукупність коду та даних, створену для відтворення властивостей фізичних предметів або абстрактних понять. Він ефективний як елемент програмування, якщо є прямою абстракцією звичайних предметів та у значній мірі приховує складність власної реалізації від користувача.

Початкові форми *ООП* були запроваджені ще у мові *Симула-67* : тоді були введені поняття класу та конструкцій, що підтримували успадкування об'єктів. Серед найперших були розроблені об'єкти, найтісніше пов'язані з комп'ютерами, такі як *Integer, Array* та *Stack.* Потім були мови *Smalltalk-72*, *CLU*, *Ada.* У мові *Smalltalk,* найбільш суперечливій щодо застосування нових форм, взагалі все було представлено у вигляді об'єктів. Решта, не менш відомі, являють собою розширення раніше існуючих мов – *Сі, Лісп*а, об'єктного *Паскалю* тощо. І тут слід виокремити властивості мови Сі++ – вони такі, що спроможні забезпечити підтримку декількох парадигм програмування.

2.2 Відмінності мов Сі та Сі++, не пов'язані з використанням об'єктів

Порівняльну характеристику Сі та Сі++ розпочнемо переліком нових ключових слів.

2.2.1 Ключові слова

Перелік ключових слів мови Сі++ дещо розширений, порівняно з Сі. До нього додано такі ключові слова :

catch	***inline***	***protected***	***throw***
class	***new***	***public***	***try***
delete	***operator***	***templeate***	***virtual***
friend	***private***	***this***	

2.2.2 Область опису змінних

У мові Сі локальні змінні повинні описуватися на початку блоку, аби їх опис передував операторам виконання. Сі++ дозволяє описувати змінні у будь-якому місці програми, у безпосередній близькості від коду, що їх використовує. Цей метод оголошення змінних зменшує вірогідність помилок, пов'язаних з їх використанням. Так, лічильник циклу може визначатися та використовуватися у самому операторі, і після цього, за межами блоку, він також залишиться доступним.

Наприклад:

```
float s=0;
for (int i=0;i<10;i++)
{
   s=s+1.0/i;
}
printf("i==%d \n s==%f",i,s);
```

Але в Сі все ще залишаються жорсткими вимоги щодо так званого блоку опису змінних на початку функції.

2.2.3 Використання коментарів

У Сі++ існує дві форми коментарів. Перша форма запозичена від Сі. Згідно неї текст повинен розміщуватися між „дужками" /* та */ (див розділ 1.2.4). Друга ж форма передбачає розміщення коментарю після двох символів //. Такий коментар закінчується переходом на наступний рядок.

```
// це новий формат коментарів в Сі++
```

Старий формат коментарів використовується здебільшого для організації багаторядкових коментарів:

```
/* багаторядковий
    коментар ...
*/
```

2.2.4 Аргументи по замовчуванню

У Сі++ з'являється нове застосування аргументів у викликах функцій - використання аргументів по замовчуванню. Таку назву має той аргумент, який можна не вказувати у викликах функцій, тоді як компілятор за необхідністю приєднає його автоматично, присвоюючи значення по замовчуванню. Аргументи по замовчуванню визначаються, як і решта, у прототипі функції. У випадку, коли є бажання передати своє значення, слід вказати його явно, таким чином перекриваючи значення по замовчуванню. Синтаксис Сі++ потребує при оголошенні та використанні аргументів по замовчуванню дотримання наступних правил:

- для використання значення аргументу по замовчуванню в якості параметру слід пропустити аргумент на місці відповідного параметру у виклику функції;
- аргументи, що приймаються по замовчуванню, розподіляють список параметрів функції на дві частини: перша, що містить параметри (цей список може бути порожнім, якщо по замовчуванню присвоєно аргументи усім параметрам); друга містить параметри, якщо присвоєно аргументи по замовчуванню;
- якщо приймається параметр по замовчуванню, слід призначити по замовчуванню аргументи й решти параметрів, що слідують далі.

Для прикладу розглянемо наступне оголошення функції:

```
int fn(int base, int count=2,int code=3);
```

Тепер організуємо виклик цієї функції наступним чином:

```
    int x=25;
1). int a=fn(x);
2). int b=fn(x,5);
3). int c=fn(x,7,25);
```

У першому виклику використовується лише перший аргумент, а решта параметрів *count* та *code* приймаються по замовчуванню. Наступний виклик функції містить вже два аргументи, перекриваючи значення по замовчуванню для аргументу *count*. Останній виклик не містить аргументів по замовчуванню взагалі, використовуючи

конкретні значення. Зауважимо, що слід розміщувати параметри для аргументів по замовчуванню у порядку збільшення вірогідності їхнього використання, аби забезпечити найбільш доцільне їх застосування на практиці. Крім того, Сі++ допускає, аби значення аргументу функції, що використовується по замовчуванню, необов'язково було константою: воно може бути глобальною змінною або навіть значенням, що повертається деякою функцією.

2.2.5 Перевантаження функцій

Згадаємо, що в Сі кожна функція повинна мати унікальне ім'я. У Сі++ є можливість визначати функції з однаковими іменами, але з унікальними типами аргументів. Одне й те саме перевантажене (*overloading*) ім'я може описувати одні й ті ж дії, що можуть проводитися над даними різних типів. Компілятор відрізнятиме одну функцію від іншої згідно вказаного прототипу, що містить число, порядок слідування та тип параметрів. Так, функції носять ім'я перевантажених, оскільки їх імена однакові, а робота, яку вони виконують, відмінна. Імена функцій можуть бути перевантажені лише в межах однієї й тієї ж області видимості. У більш ранніх версіях Сі++ на початку області видимості, в якій виконувалось перевантаження, обов'язковим було ключове слово *overload*. Починаючи з версії 2.0 необхідність у цьому відпала, хоча здебільшого для сумісності версій ключове слово залишилось, але використання його зовсім небажане.

Отже, перевантажені функції повинні відрізнятися одна від іншої принаймні за одним з нижче наведених пунктів:

мати різну кількість аргументів;

мати відмінність хоча б в одному з типів аргументів.

Приклад. Визначимо перевантажені функції S*how(),* що друкують власні аргументи. Зазначимо, що вони знаходяться в одному файлі (мають однакову область видимості - файл).

```
void Show(long value)
{
  printf("%ld",value);
}
void Show(char *string)
{
  puts(string);
}
void Show(double value)
{   printf("%lf",value);}
```

```
void main(){
  Show("\n Друк 1");
  Show(9998);
  Show(1.223);
}
```

Коли компілятор зустрічає у функції *main()* різні звертання до функції *Show()*, тобто виклики функцій з різним списком параметрів, він їх аналізує та відшукує функцію, що підходить для використання. Слід зазначити, що повністю уся додаткова робота виконується ще під час компіляції, що вельми важливо: під час виконання програми ніяких додаткових процесів не відбудеться і звертання до перевантаженої функції нічим не відрізнятиметься від аналогічного процесу звертання до будь-якої функції.

Процес пошуку функції, що найбільше підходить для використання полягає в пошуку „найкращої" відповідності формальних і фактичних параметрів. Це досягається шляхом перевірки набору критеріїв в наступному порядку:

- Точна відповідність типів; тобто повна відповідність, що досягається тривіальним перетворенням типів (наприклад, ім'я масиву і покажчик, ім'я функції і покажчик на функцію, тип *type* і *const type*).
- Відповідність, що досягається „підтягуванням" типів (наприклад, *char* в *int*, *short* в *int*, *float* в *double* тощо).
- Відповідність, що досягається шляхом стандартних перетворень (наприклад, *int* в *double*, *double* в *int*, *double* в *long double* тощо).
- Відповідність, що досягається перетвореннями, визначеними користувачем.
- Відповідність за рахунок змінного числа аргументів (...) в оголошенні функції.

Зауваження. Функції не розрізнятимуться, якщо мають лише різний тип значення, що повертається. Наприклад, не можуть вважатися перевантаженими наступні функції:

```
int Show(int *a);
char *Show(int *a);
```

Недостатнім є застосування функцій з аргументами типу посилання:

```
int Show(int a);
int Show(int &b);
```

Із зрозумілих причин неможливим є перевантаження функцій, в яких відсутній опис аргументів. Так, представлення *int proc(int a, ...);*

та *int proc(int a, char b);* не визначить, яка функція буде активізованою, якщо виклик організувати як *proc(1,2).* Не можуть бути перевантаженими функції, якщо їх параметри відрізняються лише застосуванням модифікаторів їх аргументів - *const*, *volatile*. Компілятор не розрізнятиме описи *void funk(const int a)* та *void funk(volatile int a)*. Крім того, неможливим є перевантажування усіх функцій із стандартної бібліотеки *Standard C Library*.

Компілятор Сі++ здійснює декорування імен перевантажених функцій, що дозволяє без ускладнень забезпечити використання механізму перевантаження. Для власного внутрішнього представлення він додає до імені декілька символів, що ототожнюють тип та порядок слідування параметрів, що сприймаються функцією. Як приклад, розглянемо декілька варіантів декорування:

void func(int i); → @func$qi
int func(int i); → @func$qi
void func(char i); → @func$qc
void func(char *p;) → @func$qpc

З перших рядків добре видно, чому неможливим є перевантаження функцій, що відрізняються лише типом значення, яке повертається. Адже цей тип не включається до закодованих імен.

Приклад.

```
#include<stdio.h>
#include<conio.h>
void print(int a)
{
   printf("void print(int a)\n");
}
void print(const char* a)
{
  printf("void print(const char* a)\n");
}
void print(double a)
{
  printf("void print(double a)\n");
}
void print(long a)
{
  printf("void print(long a)\n");}
void print(char a)
{
  printf("void print(char a)\n");
}
```

```
void fn(char c,int i,short s,float f)
{
  print(c);   /* точна відповідність – print(char a) */
  print(i);   /* точна відповідність – print(int a) */
  print(s);   /* "підтягування" типу – print(int a) */
  print(f);   /* "підтягування" типу - print(double a)
*/
  print('a'); /* точна відповідність – print(char a) */
  print(49);  /* точна відповідність – print(int a) */
  print(0);   /* точна відповідність – print(int a) */
  print("a"); /* точна відповідність – print(const
char* a) */
}
void main()
{
  fn(1,2,3,4.5);
}
```

Отримаємо такі результати :

```
void print(char a)
void print(int a)
void print(int a)
void print(double a)
void print(char a)
void print(int a)
void print(int a)
void print(const char* a)
```

2.2.6 Операція розв'язання видимості

Операція розв'язання видимості :: (*scope resolution operator*) дозволяє здійснити доступ до глобальної змінної чи функції з блоку, в якому оголошена локальна змінна з тим самим ім'ям. Наприклад, вираз ::*I* означає глобальну змінну *I*, навіть якщо в даному блоці оголошена локальна змінна з таким самим ім'ям *I*. Використання операції :: демонструє наступний приклад :

```
int i;    // опис глобальної змінної i
...
void main(void)
{
  int i=3; // опис локальної змінної i
  ::i = 4;// присвоювання значення глобальній змінній i
  printf ("%d\n",i); // виведення локальної i
printf ("%d\n",::i); // виведення глобальної i
}
```

2.2.7 Використання inline-специфікатору

Даний специфікатор дозволяє ввести функції з новим ключовим словом *inline*, щоб компілятор розмістив код цієї функції безпосередньо у місце виклику функції. Згадаємо, що під час компіляції звичайної функції її код розміщується у деяку область пам'яті. Файл виконання являє собою набір фрагментів коду, до яких звертається основна програма, організовуючи виклик тієї чи іншої функції. Звичайно, при кожній активації функції необхідно зберігати поточну адресу програми, аби знати, куди повернути керування після виконання функції. Не складно припустити, що програма працюватиме повільніше, втрачаючи час на виклик функції. Тому не дивує позиція деяких програмістів принципово не писати малих функцій, включаючи їх безпосередньо до програми, турбуючись перш за все про менший час виконання. Проте програма губить свої переваги, модульність зокрема. Тут у нагоді стають саме *inline*-функції, що дозволяють зберегти високомодульний стиль проектування без надлишкових витрат часу на виклики функцій. Виклик такої функції організується таким чином, що передача керування функції (команда переходу) заміняється тілом самої функції. Звичайно, що така функція повинна бути визначена до її першого використання у програмі (одного лише прототипу недостатньо), то ж найдоцільніше їх визначати у *header*-файлі.

Приклад:

```
inline int max(int a, int b)
{
   return (a>b)?a:b;
}
```

Бажаний максимальний розмір *inline*-функцій - три оператори виконання. Якщо функція займатиме більше рядків, час її виклику можливо порівняти з часом, необхідним для звичайного виконання функції. Тобто із збільшенням розміру функції переваги такої підстановки зменшуються.

Inline-функція виконується як один рядок, незалежно від того, скільки рядків вона містить та до скількох помилок призведе її виконання. Тому для проведення відлагодження тимчасово слід усунути ключове слово, аби повністю впевнитися у відсутності помилок.

Не усі функції можливо оголосити *inline* - функціями. Включення у функцію будь-якого оператора циклу призведе до перетворення у

outline-функцію (тобто звичайну, що не підставлятиметься). У цьому випадку час на виконання циклу "затьмарить" переваги оголошення *inline*. Насамкінець слід зауважити, що даний специфікатор лише вказує компілятору на необхідність оптимізації виклику даної функції підстановкою її тіла на відміну від звичайної активації функції. Підкреслимо, що тільки вказує, отже інколи компілятор спроможний ігнорувати такі вказівки. Компілятор *Visual Сі++* має ключ, що дозволяє програмісту або перетворити функцію в *outline*, якщо вона має відповідні вади (наявність керуючих структур), або залишати її як *inline*, доки є надія здобути принаймні хоч якусь ефективність алгоритму.

2.2.8 Анонімні об'єднання

Анонімні об'єднання *union* не мають імені позначення; звертання до елементів цих структур проводиться безпосередньо, подібно звичайним змінним. Згадаємо, що елементи *union* розділяють між собою одну й ту ж область пам'яті. Глобальні анонімні об'єднання повинні визначатися як статичні.

Анонімні об'єднання визначають об'єкт, а не тип. Імена членів анонімного об'єднання повинні відрізнятися від інших імен з однієї області видимості, використовуються безпосередньо, без операції "крапка".

Приклад.

```
#include<string.h>
static union { //глобальне анонімне об'єднання
   char name[80];
   long num1;
};
int main() {
   union { // локальне анонімне об'єднання
     int num2;
     int sum;
   };
   for (num2=0; num2<10; num2++)
       num1=num2;
   return 0;
}
```

Головна ідея застосування таких об'єднань - трактування та використання одного й того ж місця у пам'яті для змінних різного типу.

2.2.9 Оператори розподілу пам'яті

Для керування розподілом динамічної пам'яті в Сі++ широко використовуються оператори ***new*** та ***delete***. Вони замінюють відомі нам з мови Сі ***malloc***, ***calloc***, ***free***, проте це не означає, що всі вони не можуть використовуватися у Сі++. Уся справа саме у гнучкості нових операторів: *new* повертає покажчик на тип, для якого виділяється пам'ять, в той час як *malloc* повертає порожній покажчик, тому у першому випадку відпадає необхідність використовувати перетворення типу. Крім того, гнучкість нових операторів очевидна саме при використанні класів, оскільки клас може визначити власний варіант цих операцій (наприклад, у процедурах ініціалізації та очистки), про що йтиметься далі.

Отже, оператор *new(type)* повертає покажчик на тип *type*, для якого виділяється пам'ять. Наступні рядки використовують цей оператор:

```
float *p=new float;
double *array=new float[10];
```

Оператор *delete(var)*, навпаки, звільняє відповідну ділянку динамічної пам'яті. Застосування його до попередньо визначених змінних матиме вигляд:

```
delete p;
delete array; // або delete[10] array;
```

Примітка. Не слід викликати вперемішку функції розподілу пам'яті *ANSI C* та *C++* - у багатьох випадках робота програми завершується аварійно.

2.3 Порівняння функціонального та об'єктного підходу

Згадаємо програмну реалізацію концепції дати з використанням функціонального підходу (див. Розділ 1.13.2. „Масиви структур"). Тоді для представлення дати ми створили відповідну структуру *Data* та окремо описали набір відповідних функцій, які будуть виконувати операції з типом даних *Data*.

```
typedef struct Date
{
  int d; /* день */
  int m; /* місяць */
  int y; /* рік */
} Date;

void print_date(Date &d);
void init_date(Date &d,int dd,int mm,int yy);
int leapyear(int yy);
void add_year(Date &d,int yy);
void add_month(Date &d,int mm);
void add_day(Date &d,int dd);
```

Як видно, при використанні функціонального підходу немає явного зв'язка між типом даних і функціями. Дані відіграють пасивну роль по відношенню до функцій, тому що виникає необхідність передачі в кожну функцію змінної структури типу *Data*. У головній програмі слід забезпечити виклики усіх функцій відповідними фактичними параметрами:

```
void main(void)
{
   Date date1,date2;    /* оголошення змінних
типу Data*/
   init_date(date1,15,12,2002); /* ініціалізація
змінної date1*/
   add_day(date1,16);   /* додаємо до date1 16 днів */
   print_date(date1);   /* виводимо на екран значення
date1 */
   init_date(date2,1,1,2003); /* ініціалізація змінної
date2*/
   add_month(date2,10); /* додаємо до date1 10 місяців
*/
   print_date(date2);   /* виводимо на екран значення
date2 */
}
```

Тепер розглянемо альтернативний - ***об'єктний підхід*** до організації даних задачі. Нашою ціллю є створення програмного аналогу сутності деякого об'єкту під назвою дата, який, окрім звичайних властивостей, як день, місяць, рік, буде містити методи (дії), які можна виконувати над цим об'єктом, тобто встановлювати та змінювати дату, визначати, чи є даний рік високосним тощо. Виявляється, у мові Сі++ можна встановити тісний зв'язок між даними та функціями, що їх обробляють, оголосивши останні в якості окремих

елементів структури таким чином, що новий структурований тип набуде нових, активних властивостей:

```
struct Date
{
  int d; // день
  int m; // місяць
  int y; // рік
  void print();
  Date(int dd,int mm,int yy);
  int leapyear();
  void add_day(int dd);
  void add_month(int mm);
  void add_year(int yy);
};
```

Фактично у структуру *Date* ми додали ті ж самі функції, змінивши число їхніх параметрів, зробивши їх активними елементами структури! Функції, що оголошені всередині структури, будуть називатися функціями-членами і їх можна викликати тільки для змінної відповідного типу, використовуючи стандартний метод доступу до членів структури. В такому випадку відпадає необхідність передачі змінних типу *Data* у головній функції, так як дані та функції об'єднані в єдиному об'єкті структурного типу *Data*:

```
void main(void)
{
 Date date1(15,12,2002),date2(1,1,2003);
 /* ініціалізація date1 та date2 */
 date1.add_day(16); /* додаємо до data1 16 днів*/
 date1.print();  /* виводимо на екран значення date1 */
 date2.add_month(10); /* додаємо до data2 10 місяців */
 date2.print();  /* виводимо на екран значення date2 */
}
```

При реалізації функції можна використовувати члени даної структури без явної вказівки імені об'єкта. Наводимо програму, в якій реалізується концепція дати з використанням об'єктно-орієнтованого підходу.

```
#include<stdio.h>
#include<conio.h>
struct Date
{
  int d; // день
  int m; // місяць
  int y; // рік
  void print();
```

```
  Date(int dd,int mm,int yy);
  int leapyear();
  void add_day(int dd);
  void add_month(int mm);
  void add_year(int yy);
};

void Date::print()
/* виведення на екран дати */
{
   printf("%d.%d.%d\n",d,m,y);
}

Date::Date(int dd,int mm,int yy)
/* ініціалізація структури типу Date */
{
  d=dd;
  m=mm;
  y=yy;
}

int Date::leapyear()
/* визначення, чи високосний рік */
{
 if ((y%4==0&&y%100!=0)||(y%400==0)) return 1;
     else return 0;
}
void Date::add_year(int yy)
/* додати yy років до дати */
{
  y+=yy;
}

void Date::add_month(int mm)
/* додати mm місяців до дати */
{
  m+=mm;
  if (m>12)
  {
     y+=m/12;
     m=m%12;
  }
}

void Date::add_day(int dd)
/* додати dd днів до дати */
{
 int days[]={31,28,31,30,31,30,31,31,30,31,30,31};
```

```
 d+=dd;
 if (leapyear()) days[1]=29;
 while ((d>days[m-1]))
 {
   if (leapyear()) days[1]=29;
      else days[1]=28;
   d-=days[m-1];
   m++;
   if (m>12)
   {
     y+=m%12;
     m=m/12;
   }
 }
}

void main(void)
{
  Date date1(15,12,2002),date2(1,1,2003);
  date1.add_day(16);
  date1.print();
  date2.add_month(10);
  date2.print();
}
```

2.4 Об’єктно - орієнтоване програмування та його головні принципи

У попередньому розділі ми розв'язали задачу за допомогою нового стилю проектування програм - об'єктного. Кожний стиль програмування має свою концептуальну основу, вимагає різного підходу до розв’язування задачі. Для об’єктно-орієнтованого стилю концептуальна основа полягає в об’єктному підході. Цьому підходу відповідають чотири головних елементи: ***абстрагування, обмеження доступу, модульність*** та ***ієрархія***. Ці елементи є головними у тому розумінні, що за одним з класиків об'єктно-орієнтованого проектування програм Граді Бучем [6] без будь-якого з них підхід не буде повністю об’єктно-орієнтованим. А відсутність відповідної концептуальної основи призведе до того, що програми, які написані на мовах *Object Pascal*, *Ci++,* будуть мало відрізнятися за своєю структурою від програм відповідно на *Pascal* або *Ci*. Виразні можливості цих об’єктно-орієнтованих мов будуть або втрачені, або суттєво викривлені. Але ще більш важливим є те, що при цьому буде мало шансів впоратися із складністю розв’язування задачі.

Надамо визначення основних принципів об'єктного підходу :

1. ***Абстрагування*** – виділення таких вагомих характеристик об'єктів, які відрізняють його від усіх інших об'єктів і які чітко визначають особливості даного об'єкта з точки зору подальшого аналізу.
2. ***Обмеження доступу*** – процес захисту окремих елементів, який не впливає на вагомі характеристики об'єкта, як цілого.
3. ***Модульність*** – властивість системи, яка зв'язана з можливістю декомпозиції на ряд тісно зв'язаних модулів (частин).
4. ***Ієрархія -*** впорядкування за деякими правилами об'єктів системи.

Ідея класів – це основа об'єктно-орієнтованого програмування (ООП). Мета ООП – намагання зв'язати дані й функції для їх обробки в єдине ціле – клас. В класах об'єднуються структури даних і функції їх обробки. Ідея класів відображає будову об'єктів реального світу – оскільки кожний предмет або процес має свої властивості, будову, поведінку.

Клас – це визначений користувачем тип даних. В класі задаються властивості і поведінка якого-небудь, об'єкта у вигляді полів-даних і функцій для роботи з ними.

Парадигма програмування – це набір теорій, методів, стандартів, які використовуються при розробці та реалізації програм на комп'ютері. ООП часто називають новою парадигмою програмування, хоча її революційний поступ розпочався давно у минулому. ООП основане на трьох принципах, що надають класам нові властивості:

1. *Інкапсуляція* – об'єднання в єдине ціле даних і алгоритмів обробки цих даних. В ООП дані називаються полями, а алгоритми – методами або функціями-членами (*methods*, *member functions*).
2. *Успадкування* - властивість створення ієрархії класів, коли нащадки отримують від попередника поля і методи.
3. *Поліморфізм* (від гр. *poly* – багато і *morphos* форма, означає багато форм) – це властивість класів однієї ієрархії вирішувати схожі за змістом завдання за допомогою різних алгоритмів.

2.4.1 Абстрагування

Абстрагування - один із головних засобів, що використовуються для розв'язання складних задач.

Абстракція - це досить суттєві характеристики деякого об'єкта, які відрізняють його від усіх інших видів об'єктів і, таким чином, чітко визначають особливості даного об'єкта з точки зору подальшого розгляду та аналізу.

Абстрагування концентрує увагу на зовнішніх особливостях об'єкта і дозволяє відокремити найбільш суттєві особливості поведінки від деталей їх здійснення. Такий розподіл можна назвати бар'єром абстракції, який ґрунтується на принципі мінімізації зв'язків, коли інтерфейс об'єкта містить тільки суттєві аспекти поведінки. Корисним є ще один допоміжний принцип, який називається принципом найменшої виразності, за яким абстракція повинна охоплювати лише саму суть об'єкта, не більше, але й не менше.

Вибір достатньої множини абстракцій для заданої предметної області є головною проблемою об'єктно-орієнтованого проектування. Існує цілий спектр абстракцій, який починається з об'єктів, що приблизно відповідають сутності предметної області, та закінчується об'єктами, які не мають реальних аналогів у житті. Підвищити ступінь абстракції можна :

1. описом власних типів даних;
2. використанням функцій;
3. об'єднання типів даних і функцій у модулі;
4. використанням класів.

Найбільш цікаві для нас абстракції сутності об'єктів, тому що вони відповідають словнику предметної області. Опис поведінки об'єкта містить опис операцій, які можуть виконуватись над ним, та операцій, які сам об'єкт виконує над іншими об'єктами. Такий підхід концентрує увагу на зовнішніх особливостях об'єкта.

Повний набір операцій, які об'єкт може здійснювати над іншим об'єктом, називається протоколом (про це йтиметься нижче). Протокол відображає всі дії, які може зазнавати сам об'єкт і за допомогою яких може впливати на інші об'єкти, чим повністю визначає зовнішню поведінку абстракції із статистичної та динамічної точки зору.

2.4.2 Обмеження доступу

Створенню абстракції будь-якого об'єкта повинні передувати певні рішення про засіб її реалізації. Вибраний спосіб реалізації повинен бути схований та захищений для більшості об'єктів-користувачів (які звертаються до даної абстракції). Поняття обмеження доступу можна визначити таким чином:

Обмеження доступу - це процес захисту окремих елементів об'єкта, що не порушує суттєвих характеристик об'єкта як цілого.

Абстрагування та обмеження доступу являються взаємодоповнюючими факторами: абстрагування фокусує увагу на зовнішніх особливостях об'єкта, а обмеження доступу - або інакше захист інформації - не дозволяє об'єктам-користувачам розрізняти внутрішню будову об'єкта. Обмеження доступу, таким чином, визначає видимі бар'єри між різними абстракціями. Аналогічним чином при проектуванні баз даних програмісти не звертають уваги на фізичний зміст даних, а зосереджуються на схемі, яка відображає логічну будову даних.

На практиці здійснюється захист як структури об'єкта, так і реалізації його методів. У мові Сі++ керування доступом та видимістю досягається з великою гнучкістю. Елементи об'єкта можуть бути віднесені до загальнодоступної, відокремленої або захищеної частини (про це йтиметься нижче). Загальнодоступна частина "видима" для всіх об'єктів; відокремлена частина повністю прихована для інших об'єктів; захищена частина "видима" тільки для даного класу та його підкласів.

2.4.3 Модульність

Модульність - це властивість програмної системи, що пов'язана із можливістю декомпозиції її на ряд внутрішньо зв'язаних на слабко пов'язаних між собою модулів.

Розподіл програми на уривки (частини) дозволяє частково зменшити її складність, однак значно важливішим є той факт, що цей процес покращує опрацювання її частин. Ці частини дуже цінні для вичерпного розуміння програми в цілому. Модульність є елементом конструкції в Сі та дозволяє здійснювати на її основі проектні рішення. В мові Сі++ класи та об'єкти складають логічну структуру системи; як абстракції організуються в модулі, які утворюють фізичну структуру системи. Така властивість стає особливо корисною, коли система складається із багатьох десятків, а то й сотень класів.

В мовах, які підтримують принцип модульності як самостійну концепцію, реалізується розділ інтерфейсної частини та реалізації. Таким чином, модульність та обмеження доступу ідуть невід'ємно один від одного. В Сі модульність реалізується особливими прийомами: модулями є файли, що компілюються окремо. Для мов *Сі/Сі++* традиційним є розміщення інтерфейсної частини модулів в окремих файлах із розширенням **.h* (т. зв. *заголовочні файли*), тоді як реалізація модуля описується у звичайних файлах із розширенням **.c* або **.cpp*. Взаємозв'язок файлів реалізується шляхом підключення директиви *#include*. Такий підхід будується виключно на погодженнях та не є суворою вимогою мови, проте залишається бажаним.

Правильний розподіл програми на модулі є майже такою самою складною задачею, як визначення правильного набору абстракцій. Найчастіше модулі виконують роль певних фізичних контейнерів, в які поміщуються визначення класів та об'єктів при логічному проектуванні системи.

Відсутність стандартизованих фрагментів дає програмісту значно більшу ступінь вільності. Та в процесі розподілу системи на модулі можуть бути корисними два правила. Перше полягає у наступному: оскільки модулі є елементарними та неподільними блоками програми, що можуть використовуватися в системі багаторазово, розподіл класів та об'єктів повинен створювати для цього максимальні вигоди. Друге правило випливає із того факту, що більшість компіляторів створюють окремий сегмент коду для кожного з модулів, тому його розмір відповідно обмежений. Оголошення функцій у модулі може справляти серйозний вплив на можливість їх виклику з іншого модуля (сторінкова стратегія організації пам'яті). Велика кількість викликів між сегментами завантажує кеш-пам'ять та впливатиме на зниження характеристик системи в цілому.

Таким чином, принципи абстрагування, обмеження доступу та модульності є взаємодоповнюючими. Об'єкт визначає межі певних абстракцій, а обмеження доступу та модульність створюють бар'єри між ними.

2.4.4 Ієрархія

Абстракція - річ необхідна та корисна, проте завжди, крім найпростіших ситуацій, число абстракцій в системі набагато перебільшує можливості їх одночасного контролю. Обмеження доступу дозволяє в деякій мірі зняти цю перешкоду, усунувши з поля зору

внутрішній зміст абстракцій. Модульність також спрощує завдання, об'єднуючи логічно пов'язані абстракції в групи. Але цього, виявляється, ще недостатньо. Значне спрощення в розумінні складних задач досягається за рахунок утворення *ієрархічної* структури саме з абстракцій. Визначимо ієрархію наступним чином:

Ієрархія – це аранжована та упорядкована система абстракцій.

Основними видами ієрархічних структур стосовно до складних систем є структура класів (ієрархія за номенклатурою) та структура об'єктів (ієрархія за складом). Принципи абстрагування, обмеження доступу та ієрархії конкурують між собою - якщо абстрагування даних полягає у встановленні жорстких меж, що захищають стан та функції об'єкта, то принцип успадкування вимагає відкрити доступ і до стану, і до функцій об'єкта для майбутніх похідних об'єктів. Для будь-якого класу може існувати два види об'єктів-користувачів: "рідні" об'єкти, які використовують операції даного класу для доступу до його елементів, та об'єкти-підкласи, що отримані за допомогою успадкування даного класу. Існує три способи порушення механізму обмеження доступу через механізм успадкування: підклас може отримати доступ до даних свого суперкласу, здійснити виклик відокремленої (захищеної) функції суперкласу та звернутися напрямки до суперкласу. Різні мови програмування по-різному реалізують такі механізми успадкування та обмеження доступу, та найбільш гнучким та одночасно непростим у цьому відношенні є Сі++, про що і йтиметься пізніше.

2.5 Класи

Як було показано у розділі 2.3, структури в Сі++ дозволяють групувати в одному типі декілька елементів даних та функцій, що їх обробляють. Нижче введемо класичне поняття класу в Сі++, якому також властиві ці особливості - він може містити в собі як елементи-дані, так і елементи-функції, що спроможні обробляти ці дані.

Клас (***class***) - це визначений користувачем тип даних, що застосовується для опису абстрактної множини об'єктів, які пов'язані узагальненням структури та поведінки. У синтаксичному смислі клас в Сі++ дуже нагадує визначення структури в Сі, за виключенням деяких моментів. По-перше, він може містити в собі одну або декілька специфікацій доступу, що задаються як *public*, *private* або *protected*, про які йтиметься пізніше. По-друге, клас, зазвичай, може включати в себе ще й функції-методи поряд з елементами-даними. По-третє, класу найчастіше притаманні спеціальні функції - конструктор та деструктор

- відповідно для створення та знищення екземплярів класу - об'єктів. Насамкінець, у підтвердженні попереднього розділу, ключові слова *class* та *struct* в Сі++ однаково можуть використовуватися при описі як класів, так і звичайних структур.

2.5.1 Протокол опису класу

Протокол опису класу - опис ідентифікатору класу (типу) із вказівкою елементів-даних (*memeber data*) та повідомлень або елементів-функцій (*memeber function*), які об'єкт обробляє. Елементи-дані - це такі ж звичайні змінні, як елементи структури, елементи-функції - це функції, визначені в рамках класу, що можуть працювати лише з елементами-даними цього класу. Усі оголошення даних та повідомлень повинні знаходитися всередині протоколу класу (оголошення):

Синтаксис:

```
class <ім'я> {
    [ private : ]
    [ <опис прихованих елементів> ]
    [ protected :
      <опис захищених елементів> ]
    [ public :
    <опис доступних елементів> ]
};
```

Приклад опису класу *point* (представлення точки на площині) за допомогою *struct* :

```
struct point
{  // опис класу за допомогою struct
   private:     // специфікатор доступу
     int x,y;      // приховані елементи-дані
   public:      // специфікатор доступу
     void setx(int x);// далі–відкриті елементи-функції
     void sety(int y);
     int getx();
     int gety();
};
```

Наведемо приклад опису класу за допомогою *class* :

```
class line {
    // по замовчанню – private елементи
    int x1, y1, x2, y2;
   public:
    line(int x1,int y1,int x2,int y2);
    void show();
    ~line();
};
```

В даному прикладі оголошується клас для представлення відрізка на площині. В цьому класі передбачено чотири приховані елементи-змінні *x1,y1,x2,y2*, конструктор *line(int x1,int y1,int x2,int y2);* , деструктор *~line()* та метод *void show();*.

Будь-яка змінна, оголошена (визначена) у класі, має область видимості класу (*class scope*), що простягається з місця її опису до закінчення протокольного опису класу. Даними-членами класу можуть бути змінні будь-якого типу, включаючи інші класи, покажчики на типи об'єктів класів тощо. Проте існують обмеження на використання елементів-даних.

Елементи-дані :

1. можуть мати будь-який тип, окрім типу цього ж класу (але можуть бути покажчиками або посиланнями на цей клас);
2. можуть бути описані з модифікатором *const*, при цьому вони ініціалізуються тільки один раз (за допомогою конструктора) і не можуть змінюватися;
3. можуть бути описаними з модифікатором *static*, але не як *auto*, *extern* і *register*;
4. ініціалізація полів при описі не допускається.

Функції, описані (або визначені) у протоколі класу, носять назву *функцій-членів* (*елементів-функцій*), щоб відрізняти їх від звичайних зовнішніх "некласових" функцій. Визначення функцій-членів може знаходитися як всередині протоколу, так і поза оголошенням класу (у цьому або іншому файлі). Функції-члени, визначені у класі, виглядають як звичайний опис без попереднього оголошення їх прототипу.

Приклад 1 (визначення функцій-членів в протоколі опису класу):

```
class line
{
   int x1, y1, x2, y2;
  public:
   line(int _x1,int _y1,int _x2,int _y2)
   {
      x1=_x1; y1=_y1;
      x2=_x2; y2=_y2;
   }
   void show()
   {
   }
   ~line()
   {
   }
};
```

Функції-члени, визначені таким чином, є по замовчуванню *inline-функціями*, так як здебільшого вони невеликого розміру та найчастіше не містять в собі циклів. Для попередження можливої нечитабельності протоколу класу та при використанні функцій великого розміру більш раціонально розміщувати їх визначення в іншому місці (поза протоколом, у цьому або ж іншому файлі). При цьому слід обов'язково вказати прототип функції у протокольній частині.

Приклад 2 :

```
//line.h
class line {
    int x1, y1, x2, y2;
   public:
    line(int _x1, int _y1, int _x2, int _y2);
    void show();
    void move(int x,int y);
    ~line();
};

// line.cpp
#include "line.h"
line::line(int _x1,int _y1,int _x2,int _y2)
{
   x1=_x1; y1=_y1;
   x2=_x2; y2=_y2;
}
```

```
void line::move(int x,int y)
{
   x1+=x;
   y1+=y;
   x2+=x;
   y2+=y;
}
```

З вищенаведеного прикладу 1 випливає, що немає необхідності включати ім'я класу в ім'я елемента-функції при визначенні її у протокольній частині опису класу. Однак треба визнати, що поза протоколом така вказівка обов'язкова (див. приклад 2) - тут ми застосовуємо операцію розв'язання видимості **::** (*scope resolution operator*), вказуючи належність до конкретного класу (наприклад, *void line::move(int x, int y)*). Функція *move(int x, int y)* без імені класу буде звичайною зовнішньою функцією. До речі, вона може бути описана як зовнішня і таким чином – *::move(int x, int y),* коли є декілька функцій з однаковими іменами, як у наступному випадку:

```
//визначення зовнішньої функції move()
void move(int x, int y) {
         // ...
}
class line {
    int x1, y1, x2, y2;
   public:
    // визначення функції класу move()
    void move(int x,int y) {
         // ...
    }
    void set(int x, int y) {
       move(x,y);      // line::move(x,y)
       ::move(x,y);    // зовнішня ::move(x,y);
    }
};
```

2.5.2 Створення об'єктів. Доступ до полів та методів

Лише після створення змінної класу (екземпляру, об'єкту), що має тип класу, можна отримати доступ до даних та функцій, що належать класу. Оскільки елементи-дані та елементи-функції є частиною класу, звертання до них проводиться через оголошену змінну типу "клас", подібно звертанню до елементів структури:

```
class line {
   public:
     int x1, y1, x2, y2;
     line(int _x1, int _y1, int _x2, int _y2);
     void show();
     void move(int x,int y);
     ~line();
};
//… … … … … … … … … …
void main()
{
    // створюється об'єкт s класу line
    line s(12,43,33,43);
    int localx, localy;
    localx=s.x1;      //доступ до елементу-даних x1
    localy=s.y1;      //доступ до елементу-даних y1
    s.show();  //доступ до елемента-функції show();
}
```

Отже, доступ до членів класу здійснюється за допомогою оператору “крапка”(.) - *object.data*. Цей оператор повідомляє про те, що потрібно забезпечити доступ до елементу класу *data*, змінна якого являє собою окремий його екземпляр - *object*. Ясно, що виклик елемента-функції без вказівки екземпляра класу також неможливий. Синтаксис виклику такої функції нагадує суміш синтаксису доступу до елементів-даних та виклику звичайної функції - *object.funk()*. Кожний екземпляр класу має свою множину даних та функції-елементи працюють з окремим набором даних, що належить змінній.

Можливим є спосіб посилання на елемент даних об’єкту із використанням покажчика:

```
class st
{
  public:
   int h;
   float g;
   float add(int h1, float g1)
   {
     // …
   }
};
```

```
void some_fun(st *ps)
{
   ps->h=17;
   ps->g=3.00;
   ps->add(2,3);
}
st s;
int main(void)
{
   some_fun(&s);
   return 0;
}
```

Екземпляр, для якого викликається функція-елемент, є *"поточним"* та всі посилання на елементи-дані мають відношення саме до цього екземпляру, якщо не посилаються явно на інший екземпляр. Адреса екземпляру передається функції як неявний прихований аргумент, що має назву *this* (цей, поточний*)*. Цей тип завжди є покажчиком на екземпляр відповідного класу, що, зокрема, знаходить своє застосування при перевантаженні операцій. Кожного разу, коли функція-елемент посилається на елемент поточного класу без явної вказівки екземпляру, завжди вважається, що цей екземпляр *this:*

```
class demo {
   int i;
  public:
   void load_i(int val) {
      this->i=val;
   }
   void get_i()
   {
      return this->i;
   }
};
```

```
class demo {
   int i;
  public:
   void load_i(int val) {
      i=val;
   }
   void get_i()
   {
      return i;
   }
};
```

Функції-елементи можуть перевантажуватися так, як і звичайні функції. Так як ім'я класу є частиною розширеного імені функції, є змога описати одну й ту ж функцію, але в різних протоколах класів. При активації такої перевантаженої функції для її ідентифікації використовуються не тільки аргументи, але й тип об'єкту, для якого вона викликається.

2.5.3 Використання специфікаторів доступу класу

Головною турботою класу є необхідність приховувати якомога більше інформації, аби захистити його зміст від можливого стороннього впливу. Це накладає ряд обмежень на використання даних. Тому ознакою пристойного стилю програмування на *Ci++* вважається правильне застосування специфікаторів доступу до елементів-даних та елементів-функцій класу. Таблиця описує призначення специфікаторів доступу:

Таблиця 2.1. Специфікатори доступу

public	Елементи-дані та елементи-функції доступні для функцій-елементів та інших функцій, де має місце представник класу
private	Елементи-дані та елементи-функції доступні лише для функцій-елементів поточного класу
protected	Елементи-дані та елементи-функції доступні лише для функцій-елементів поточного класу та класів, похідних від нього.

Кожний з користувачів класу (сам клас, його представники або ж похідні класи) володіє різними привілеями доступу, що визначається відповідним ключовим словом (див. таблицю специфікаторів доступу). Розділи з різними привілеями доступу можуть з'являтися у будь-якому порядку та у будь-якій кількості. Якщо усі елементи-дані та елементи-функції класу оголосити приватними (*private*), то з таким класом ніяких операцій проводити неможливо: уся інформація з протоколу класу буде прихованою. Та навпаки, до всього, що оголошено у секції *public*, дозволений необмежений доступ. Щодо специфікатору *protected*, елементи-дані стають доступними у похідному класі через прихований по замовчуванню покажчик *this* :

```
class One
{
  protected:
   int a;
};

class Two : public One
{
  // визначення похідного класу
  public:
   void example()
   {
      a=0; // означає this->a=0
   }
};
```

Якщо функція-елемент прийматиме як параметр покажчик або посилання на інший об'єкт, правило дещо змінюється: неможливо звертатися до *protected* - даних через зовнішнього користувача класу *One*:

```
class One
{
  protected:
   int a;
};

class Two : public One
{
  public:
   void example(One &A)
   {
     a=0;    // означає this->a=0
     A.a=0; // помилка доступу
   }
};
```

2.5.4 Правила визначення конструкторів

Конструктор - спеціальна функція класу, що викликається автоматично для створення та ініціалізації екземпляру певного класу. Її основне призначення полягає у тому, щоб об'єкт отримав саме те значення, яке є допустимим для даного класу. Мається на увазі, що у класі визначається спеціальна функція, яку викличе програма у момент ініціалізації об'єкту (екземпляру). Відповідальність за її виклик покладається на компілятор - справа програміста лише визначити та

описати її з тим же ім'ям, що і сам клас. Наприклад, розглянемо фрагмент:

```
class cdemo
{
   long count;
  public:
   cdemo();
   void func();
};

// визначення конструктора по замовчуванню
cdemo::cdemo()
{
   printf("Створення  об'єкту cdemo");
   count=0; // ініціалізація внутрішніх даних
}
```

Конструктор присвоює змінній *count* початкове значення та видає на екран повідомлення про створення об'єкту класу *cdemo*. При створенні об'єкту вказаного типу функція-конструктор буде викликана автоматично. Це найпримітивніший конструктор.

Існує декілька правил стосовно використання конструкторів:

1. Конструктор не повертає значення, навіть типу *void*. Неможливо отримати покажчик на конструктор.
2. Клас може мати декілька конструкторів з різними параметрами для різних видів ініціалізації (при цьому використовується механізм перевантаження).
3. Конструктор, що викликається без параметрів, називається конструктором по замовчанню.
4. Параметри конструктора можуть мати будь-який тип, крім цього ж класу. Можна задавати значення параметрів по замовчанню. Їх може містити тільки один із конструкторів.
5. Конструктори не успадковуються.
6. Конструктор не може бути оголошений як *const*, *virtual*, *static* або *volatile*.
7. Конструктори глобальних об'єктів викликаються до виклику функції *main()*. Локальні об'єкти створюються, як тільки стає активною область їх дії. Конструктор запускається і при створенні тимчасового об'єкту (наприклад, при передачі об'єкта з функції).

8. Локальні та статичні об'єкти створюються в порядку розміщення їх оголошень, статичні об'єкти - лише однин раз.

9. Конструктор викликається, якщо в програмі зустрілася яка-небудь із синтаксичних конструкцій :

   ім'я_класа ім'я_об'єкта [список_параметрів] ;
   ім'я_класа (список_параметрів);
   ім'я_класа ім'я_об'єкта = вираз;

10. Конструктор не спроможний видати повідомлення про помилку під час ініціалізації, адже він не повертає значення. Для організації повідомлення про помилку з конструктора можна використовувати механізм обробки виняткових ситуацій.

2.5.5 Методи ініціалізації елементів у конструкторах.

Оголошення та використання конструкторів може проходити за одною із наступних схем:

І. Створення об'єктів з ініціалізацією по замовчуванню.

Конструктор, що оголошується без аргументів, є конструктором по замовчуванню (*default constructor*). Якщо він не визначений в описі класу, компілятор створює його по замовчуванню (на практиці він просто виділяє пам'ять при створенні об'єкту свого класу). Як приклад, можна розглянути наступне:

```
class sphere
{
  public:
   float r;
   float x,y,z;
   sphere()
   {
      x=1.0;
      y=2.0;
      z=3.0;
      r=4;
   }
};
```

ІІ. Створення об'єктів із спеціальною ініціалізацією.

Більшість конструкторів використовуються з аргументами. Адже не дуже вдалим буде рішення, після того, як створивши порожній об'єкт, потім ще додатково викликати окрему функцію ініціалізації для збереження в ньому певних даних. Наведений нижче код допоможе

ініціалізувати об'єкт саме у момент його створення. Нехай є таке оголошення конструктора у попередньо відомому класі *sphere*:

```
sphere(float xcoord, float ycoord, float zcoord,
                                          float radius);
```

Тоді визначення конструктора має бути наступним:

```
// визначення конструктора
sphere::sphere(float xcoord,float ycoord,float
zcoord,float radius)
{
   x=xcoord; y=ycoord;
   z=zcoord; r=radius;
}
```

Враховуючи таке оголошення конструктора, необхідно при створенні об'єкту передати йому аргументи, що робить ініціалізацію екземпляру схожим на виклик функції :

```
sphere s(1.0, 2.0, 3.0, 4.0);
```

Як бачимо у такому разі, змінна визначається та ініціалізується в одному рядку. Для найбільш гнучкої ситуації іноді є сенс оголосити конструктори з різним типом та кількістю аргументів, що дозволяє будувати більш зрозумілі та пристосовані до користувача класи. Тобто конструктори, як і звичайні функції-елементи, можна перевантажувати. Компілятор, проаналізувавши аргументи при створенні об'єкту, спроможний автоматично визначити, який з конструкторів викликати. При цьому використовуються відомі правила відповідності аргументів при перевантаженні функцій.

Крім того, в Сі++ зустрічаються дві форми ініціалізації з параметрами:

1) ініціалізація у тілі конструктора, відома за вищенаведеними прикладами, схематично представлена як:

```
class two{
   int x,y;
  public:
   two(int x1,int y1)
   {
     x=x1; // присвоєння у тілі конструктора
     y=y1;
   }
};
```

2) ініціалізація списком після заголовку визначення функції:

```
class one{
   int x,y;
  public:
   one(int x1,int y1) : x(x1),y(y1)
   {
   }
};
```

Обидві форми абсолютно рівнозначні, друга, правда, зустрічається дещо рідше.

III. Створення об'єктів шляхом копіювання інших об'єктів.

У цьому випадку мається на увазі отримання копій вже існуючого об'єкту, що, у свою чергу, потребує особливого конструктора, що носить назву ініціалізатора копії (*copy initializer*) або конструктор копіювання (*copy constructor*).

Конструктор копіювання – це спеціальний вид конструктора, який отримує в якості єдиного параметра покажчик на об'єкт цього ж класу. Наступний код демонструє використання такого конструктора:

```
//визначення конструктора копіювання
example:: example(example& referance)
{
   count = referance.count;
}
void main()
{
   example object(5); //використання конструктора для int
   example object1=object;//використання конструктора
         // копіювання
         //відбувається копіювання між object1 та object
}
```

Конструктори копіювання надзвичайно важливі у випадках, коли потрібно створити копію об'єкту класу. Без них компілятор не спроможний виконати копіювання - все повністю залежить від існуючого конструктора. Копіювання об'єктів відбувається при передачі об'єктів за значенням у функцію або навпаки. Розглянемо приклад, що демонструє передачу об'єктів за значенням:

```
// функція отримує змінну класу за значенням
void func(sorce obj)
{
   obj.classfunc(); // виклик компонентної функції
класу sorce
}
void main(void)
{
   sorce obj (10); // автоматична змінна
   func(obj);   //передача функції об'єкта за значенням
}
```

При активації *func (obj)* компілятор використовує конструктор копіювання для розміщення в стеку копії об'єкту в якості аргументу. Передача за значенням зовсім ще не означатиме, що функція обов'язково отримає точну побайтову копію об'єкту. Адже те, що отримає функція, повністю залежить від того, що підготовлено конструктором копіювання. Подібна до цього ситуація і тоді, коли об'єкт повертається за значенням. (Зауваження: деякі компілятори можуть працювати інакше).

Конструктори у попередніх прикладах з'являлися у секції *public*, що зустрічається найчастіше. Проте це зовсім не є обов'язковим. Приватні конструктори не допускають створення об'єктів класу звичайними користувачами та для створення об'єкту вимагають наявності певних умов:

- конструктор може викликатися статичним членом класу;
- конструктор може викликається дружнім класом;
- відповідний об'єкт класу має функцію-елемент, що викликає конструктор для створення нового об'єкту.

Примітка. Взагалі конструктори не є обов'язковою частиною протокольного опису класу. Якщо ви не визначили ніяких конструкторів, компілятор Сі++ по замовчуванню генерує конструктор "по замовчуванню" (даруйте за каламбур - той, що не має параметрів), який обнуляє усі елементи-дані поточного екземпляру.

2.5.6 Деструктори

Деструктор - це спеціальна функція, яка є доповненням конструктора, і викликається кожного разу, коли знищується представник класу. В той час, коли функція-конструктор виділяє пам'ять для створення класових змінних, функція-деструктор звільняє цю пам'ять повністю, знищуючи змінні. Він має те саме ім'я, що й

ім’я класу, але на початку обов'язково з префіксом – тільдою(~): *~ClassName();*.

Існує декілька правил застосування деструкторів:

1. Деструктор не має аргументів.
2. Деструктор не повертає значення.
3. Деструктор не успадковується.
4. Деструктор не може бути оголошений як *const, static, volatile*.
5. Деструктор може бути оголошений як *virtual*, і, найчастіше, повинен так оголошуватися, особливо у випадках, коли має місце ієрархія успадкування.

Деструктор викликається автоматично, коли об’єкт виходить з області видимості:

1. для локальних об’єктів – при виході з блоку, в якому він оголошений;
2. для глобальних – як частина процедури виходу з *main*();
3. для об’єктів, заданих через покажчики, деструктор викликається неявно при використанні операції *delete*.

Деструктор може виконувати і деякі інші дії, наприклад, виведення остаточних значень елементів-даних класу, що буває зручно при відлагодженні програми. Проте найчастіше ця функція нічого не робить, що характерно для більшості деструкторів:

```
~ClassName(){}
```

Деструктори - антиподи конструкторів. Те, що було створено конструктором, має бути знищено деструктором. Зазвичай, деструктор знищує динамічні змінні, на які посилаються елементи-дані об'єктів класу. Приміром, організація деструктора для знищення рядка, що зберігається в об'єкті класу протягом всього життєвого циклу екземпляру, може бути таким:

```
class Example
{
  private:
   char *dts;
  public:
   Example()
   {
      dts=new(char[8]);
      dts="Example";
      cout << "Creating object [" << dts << "]\n";
   }
```

```
   ~Example()   //  реалізація деструктора
   {
      cout << "Deleting object [" << dts << "]\n";
      delete dts;
   }
};

int main()
{
   Example a1;
   return 0;
}
```

С++ викликає деструктор, коли об'єкт класу виходить з області видимості, або коли він знищується. Якщо функція-деструктор не містить ніяких дій, немає особливої необхідності її створювати. Як тільки змінна типу клас припиняє існування, виділена для об’єкту пам’ять звільняється автоматично. Це є можливим тому, що у випадку, якщо ми не визначаємо деструктор у програмі, компілятор генерує деструктор по замовчуванню.

2.5.7 Порядок виклику конструкторів та деструкторів.

При класовому успадкуванні викликаються конструктори та деструктори як базового, так і похідних класів. Важливо тут зрозуміти порядок виклику цих спеціальних функцій.

При створенні класу, похідного від базового, у першу чергу виділяється область пам’яті для базового класу, що активізує його конструктор, а вслід за цим – і конструктор похідного класу. При знищенні похідного класу, навпаки, спочатку викликається деструктор похідного класу, а потім – деструктор базового класу. Але деструктору похідного класу, на відміну від конструктора цього класу, не потрібно явно викликати деструктор базового класу. Компілятор автоматично генерує виклики базових деструкторів.

2.5.8 Статичні члени класу

Члени класу, оголошені з модифікатором класу пам'яті *static*, носять назву статичних членів класу. Вони є загальними для усіх об'єктів даного класу: змінивши значення статичного члену класу в одному об'єкті, ми отримаємо змінене значення в усіх інших об'єктах. Оголошення статичних членів-даних класу в середині оголошення класу не буде одночасно описом змінних, оскільки при цьому під ці

дані пам'ять не виділятиметься. Це слід робити в програмі окремо. Таким чином, усі об'єкти класу посилатимуться на одне й те ж саме місце у пам'яті.

Функції-елементи класу також можуть бути оголошені статичними, але оскільки вони не отримують прихований покажчик *this*, вони не можуть звертатися до нестатичних членів класу. Крім того, статична функція не може бути віртуальною. Звертання до статичних функцій та елементів можливе навіть тоді, коли ще не створено жодного об'єкту класу, в якому використовуються статичні дані. Якщо функція *func()* є статичною функцією класу *A*, її можна викликати таким чином:

```
A::func();
```

Звертання до статичних функцій з боку об'єктів класів є повністю аналогічним звичайним функціональним викликам елементів-функцій. Щодо найпростішого застосування, статичні члени можна використовувати для підрахунку кількості створених або існуючих в даний момент об'єктів класу.

2.6 Успадкування

2.6.1 Механізм успадкування

Успадкування – це співвідношення між класами, коли один клас використовує структурну або функціональну частину іншого (інших) класів.

Для початку розглянемо структуру звичайного класу, що має справу з описом людей, які працюють на одній фірмі. Пропонується така структура:

```
class employee
{                        // тип - клас «службовець»
  char *name;           // ім'я
  short age;            // вік
  short department;     // відділ
  int salary;           // зарплата
  employee *next;       // покажчик на представника класу
}
```

Список однотипних службовців буде пов'язаний через поле *next*. Тепер окремо визначимо клас *manager*:

```
class  manager
{
  employee emp;     // запис про менеджера як службовця
  employee *group;
                  // підпорядковані службовці менеджера
  short level ;    // рівень та інші характеристики
}
```

Таким чином, дані, що відносяться до службовця *employee*, зберігаються в члені *emp* класу *manager*. Все ніби очевидно, однак для компілятора немає вказівки, що покажчик на менеджера *manager** є покажчиком на службовця *employee** , виходячи з контексту структури. Більше того, зовсім немає нічого такого, що виокремлює член класу *manager emp*, що є типом *employee*. Можна, звичайно, написати спеціальний код , що застосовує до *manager** явне перетворення типу, або розмістити у списку службовців адресу члена *emp* - все це заслуговує на увагу, але й на додаткові витрати. Проте коректний підхід тут полягатиме саме у застосуванні механізму успадкування, аби встановити, що менеджер є службовцем з деякою допоміжною інформацією, тобто встановити, що один з класів (похідний) в якійсь мірі містить, повторює функціональну частину іншого (базового) класу.

Синтаксис успадкування:

```
class Base{
    // протокол базового класу
};
class Derived: [public/protected/private] Base
{
    //протокол похідного класу
};
```

Отже, логіка речей підказує такий тип відношення між класами (необов'язковий при цьому специфікатор доступу при визначенні успадкування опустимо):

```
class employee // тип - клас «службовець»
{
   // протокольна частина класу employee
};

class manager: employee
{
  // протокольна частина класу manager
};
```

У нашому випадку клас *manager* є *похідним* класом від класу *employee*, а клас *employee* є *базовим* класом для класу *manager,* що можна відобразити наступною схемою:

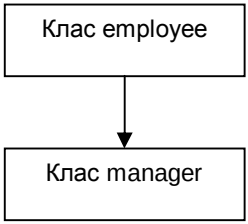


Рис. 2.1. Схема відношення успідкування між класами

У такому випадку об'єкт похідного класу додатково до власного члена *group* матиме члени класу *employee* (*name, age* и т.д.), що в принципі відповідає логіці даної задачі. Маючи визначення вищеописаних класів, можна створити список службовців, деякі з котрих є менеджерами, що наведено нижче.

```
void f()
{
    manager m1, m2;
    employee e1, e2;
    employee* elist;
    elist = &m1;   // розмістити m1, e1, m2 та e2 в elist
    m1.next = &e1;
    e1.next = &m2;
    m2.next = &e2;
}
```

Рис. 2.2. Покажчик на базовий клас (1)
та покажчик на похідний клас (2)

Звернемо увагу на таке: оскільки менеджер є службовцем, покажчик на *employee* може посилатися не тільки на об'єкт свого класу, але й на похідний об'єкт класу - *manager*. Однак службовець необов'язково є менеджером, тому використовувати *employee** замість *manager** є неможливим. Саме це відображено на рис. 2.2.

2.6.2 Керування доступом при успадкуванні

Повернемося до синтаксису успадкування. Специфікатори доступу - *public, private, protected* при призначенні типу успадкування можуть пропускатися (як, до речі, і було у нашому першому прикладі зі службовцями та менеджерами), при цьому керуються наступними правилами:

- якщо визначається *class*, то по замовчуванню похідний клас приймається як *private;*
- якщо доступ не вказаний в успадкуванні при описі *struct*, то по замовчуванню він приймається як *public*;

Наступна таблиця містить визначення рівня доступу в середині похідного класу. У першій колонці - специфікатор доступу, що визначає успадкування між класами, у двох подальших - рівень доступу у базовому та похідному класах:

Таблиця 2.2. Рівні доступу у базових та похідних класах при успадкуванні

Тип успадкування class A: [ ]class B	*Доступ у базовому класі A*	*Доступ у похідному класі B*
public	private public protected	недоступно public protected
private	private public protected	недоступно private private
protected	private public protected	недоступно protected protected

З цієї таблиці, видно, які можливості надає механізм успадкування. Так, при відкритому успадкуванні *public* загальнодоступні та захищені елементи-дані зберігають свої рівні доступу надалі, і лише *private*-елементи виявляються недоступними вниз по ієрархії.

Слід дотримуватися наступних правил успадкування методів у похідному класі:

1. Оскільки конструктори не успадковуються, похідні класи повинні мати власні конструктори. Тут можуть бути дві ситуації:
   - якщо у конструкторі похідного класу відсутній явний виклик конструктора базового класу, автоматично викликається конструктор базового класу по замовчуванню (той, що не має параметрів). Для ієрархії декількох рівнів конструктори базових класів викликаються, починаючи з найвищого рівня.
   - якщо конструктор базового класу потребує вказівку параметрів, він повинен бути явно викликаний в конструкторі похідного класу списком ініціалізації (див. розділ 2.5.5. "Методи ініціалізації елементів у конструкторах").
2. Оскільки деструктор не успадковуються та програмою не визначений деструктор у похідному класі, його буде згенеровано по замовчуванню і через нього викликано деструктори усіх базових класів. У класовій ієрархії деструктори викликаються у порядку, зворотному до виклику конструкторів; спочатку деструктор поточного класу, а потім деструктор базового класу.
3. Похідний клас може перевизначати метод з одним і тим же ім’ям, що і у базовому класі, відповідно коректуючи його поведінку для себе. Аби запобігти неоднозначностям, рекомендовано перевизначати лише віртуальні методи класів (див. розділ 2.7 "Поліморфізм").

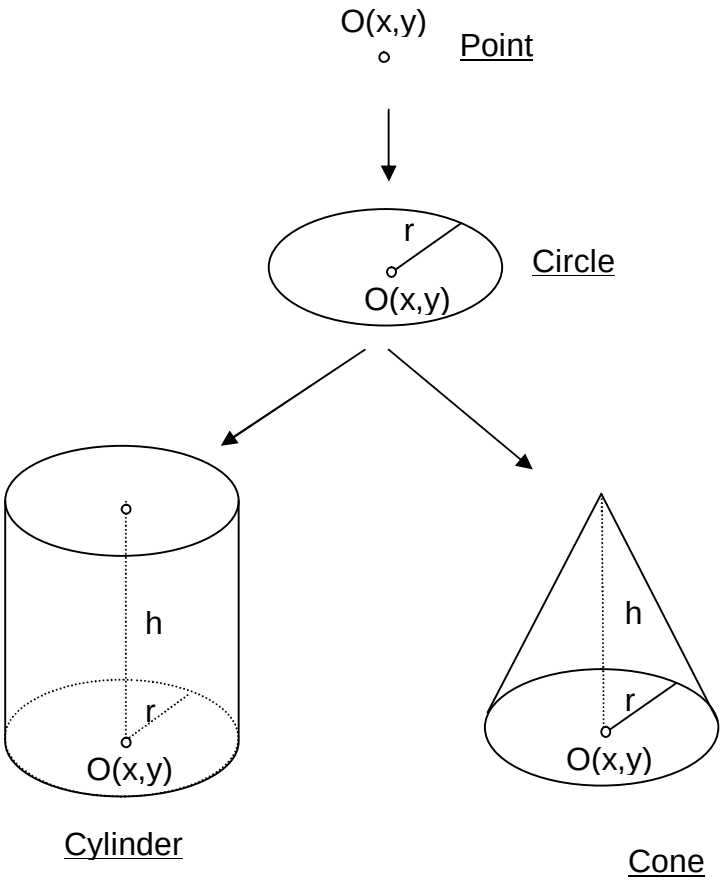


Рис. 2.3. Приклад простого успадкування

Надалі розглянемо приклад простого успадкування. Від класу точки *Point* утворимо клас кола *Circle*, а від нього - похідні класи - *Cone* (конус) та *Cylinder* (циліндр).

Нижче наведений код демонструє взаємозв'язок елементів-даних та елементів-функцій базового та похідного класів.

```
/* приклад розробки базового та похідних класів */
#include<stdio.h>
#define pi 3.1415926
// протоколи класів
```

```
class Point
{
   protected:
    float x,y;
   public:
    Point(float _x1,float _y1);
};
class Circle:public Point
{
   protected:
    float r;
   public:
    Circle(float _x,float _y,float _r);
    float get_s();
};
class Cilinder:public Circle
{
    float h;
   public:
    Cilinder(int _x,int _y,int _r,int _h);
    float get_v();
};
class Cone:public Circle
{
    float h;
   public:
    Cone(int _x,int _y,int _r,int _h);
    float get_v();
}
//реалізація функцій класів
Point::Point(float _x,float _y)
{
    x=_x;
    y=_y;
};
Circle::Circle(float _x,float _y,float _r):Point(_x,_y)
{
    r=_r;
}
Cilinder::Cilinder(int _x,int _y,int _r,int _h) :
Circle(_x,_y,_r)
{
    h=_h;
};
float Circle::get_s()
{
    return pi*r*r;
}
```

```
float Cilinder::get_v()
{
   return get_s()*h;
}
Cone::Cone(int _x,int _y,int _r,int _h):
circle(_x,_y,_r)
{
     h=_h;
}
float Cone::get_v()
{
     return h*get_s();
}
void main()
{
 Cilinder cil(1,1,10,20);
 Cone con(100,100,15,100);
 printf("cilinder:Sосн=%.2f V=%.2f\n",cil.get_s(),
                                  cil.get_v());
 printf("cone    :Sосн=%.2f V=%.2f\n",con.get_s(),
                                  con.get_v());
}
```

Слід відмітити, що у базовому класі *Point* значення *x* та *y* віднесені до захищеної частини (*protected*). Якщо їх віднести до закритої (*private)* секції класу, то у похідних класах *Circle, Cone* та *Cilinder* вони будуть недоступними.

До речі, як видно з наведеної у цьому розділі таблиці визначення рівня доступу в середині похідного класу, якщо базовий клас успадковується як *private*, його елементи типу *public* будуть *private*-елементами у похідному класі. Однак можна вибірково деякі з елементів базового класу зробити *public*-елементами у похідному класі, явно вказавши їх у секції *public* похідного класу:

```
class Base
{
   public:
    void func1();
    void func2();
};
class Base1:private Base
{
  public:
  Base::func1(); //робить void Base::func1() доступною як public
};
```

2.6.3 Друзі-класи та друзі-функції

Роль специфікаторів доступу, як відмічалося раніше, полягає в обмеженні доступу, тобто захисті інформації, що є одним з елементів об'єктного підходу. Звичайно, концепція приховування даних існує не для того, щоб її порушували, проте може статися такий випадок, коли необхідно, як виняток, забезпечити доступ конкретній функції або класу до елементів протокольної частини, специфікованими як *private* або *protected*. Семантика мови дозволяє оголосити два види таких "друзів"- один клас як єдине ціле може бути другом іншого класу, або другом може бути оголошена окрема зовнішня функція.

У тому випадку, коли не йде мова про успадкування між класами, є можливість дозволити доступ до будь-яких елементів класу іншому класу або функції за допомогою зарезервованого слова *friend*. Так, дійсно, це порушення обмеження доступу, проте іноді воно буває корисним, якщо класи не можуть бути пов'язані відношенням успадкування. Наприклад, наступний фрагмент не скомпілюється, видавши повідомлення про помилку:

```
class Room {
  private:
   double square;
   public:
   Room()
   {
      square=20.5;
   }
};
class House {
  private :
    Room object;
  public:
    void show(void)
    {
     // cannot access private member declared in class 'Room'!!!
     cout<< object.square;  //не має доступу до private-елемента!
};
```

Елемент *square* прихований в класі *Room*, і навіть заміна специфікатору на *protected* не вирішить проблеми, оскільки класи не пов'язані відношенням успадкування. Тільки віднесення *square* до *public*-частини надасть бажаного результату, проте і це не вихід: у

такому випадку й інші класи матимуть необмежений доступ до елемента-даних. Існує альтернатива, якщо оголосити клас *House* дружнім до класу *Room*:

```
class Room_ {
  friend class House;
  // далі повний протокол класу Room;
}
```

Такий опис надасть компілятору вказівку забезпечити класу *House* повний доступ до захищеної та закритої частин класу *Room* (кажуть, клас *House* є дружнім класом по відношенню до класу *Room*). Існує ряд правил відносно класів-друзів:

- Якщо клас *ONE* оголошує клас *TWO* дружнім для себе, то це не означає, що клас *ONE* буде також дружнім до класу *TWO* (не існує взаємності).
- Якщо клас *ONE* оголошує клас *TWO* дружнім для себе, класи, похідні від класу *TWO*, не будуть автоматично отримувати доступ до елементів класу *ONE*, тобто не будуть його друзями (друзі не успадковуються).
- Якщо клас *ONE* оголошує клас *TWO* дружнім для себе, класи, похідні від класу *ONE*, не будуть автоматично отримувати нові властивості, як друзі класу *TWO*.
- Специфікатори доступу не впливають на описи *friend*.

Найчастіше оголошення класів друзями є примусовим та свідчить про не досить вдалу продуманість типової ієрархії. Дійсно, якщо є необхідність у такій "співдружності", чи не було доцільним передбачати такий зв'язок на початковій стадії проектування, застосувавши механізми успадкування? У будь-якому разі, користуватися друзями в Сі++ слід лише у випадках крайньої необхідності.

Подібно дружнім класам, можливим є оголошення функцій-друзів – зовнішніх функцій або елементів-функцій іншого класу, що можуть здійснювати безпосередній доступ до всіх елементів та функцій класу, для якого вони дружні, в тому числі, оголошеними як *protected* та *private*. Правила опису та особливості дружніх функцій можна подати у такому порядку:

- дружня функція оголошується в середині класу, до елементів якого потрібен доступ, з ключовим словом *friend*. Найчастіше у якості параметра такій функції повинен передаватися сам об’єкт, або

посилання на об'єкт класу, оскільки покажчик *this* їй не передається.

- дружня функція може бути звичайною функцією або методом іншого визначеного класу. На цю функцію не поширюються дії специфікаторів доступу, не має значення конкретне місце її розташування у протоколі.
- одна функція може бути дружньою по відношенню до декількох класів.

Для початку розглянемо випадок, коли дружня функція *External()* до класу *A* є зовнішньою:

```
class One {
  private:
   char*str;
  public:
   One(){ str="123";}
   friend void Importante (One &a);
                                  //оголошення friend - функції
};
void External (One &a);
int main(void)
{
   One r;
   External(r);
   return 0;
}
void External (One &a){
cout<< a.str<<endl;   // доступ до елемента-даних
                      // char*str з секції private
}
```

Таким чином, оголошення *friend*-функції *Importante()* у класі *One* відкриває їй доступ до закритих та захищених членів даного класу. Одна й та сама функція може бути оголошеною як *friend* двом класам одночасно. Іноді це зручно, скажімо у випадках одночасної ініціалізації даних, хоча такі варіанти організації спричиняють неявний зв'язок між класами, їх подальшу залежність, що не є бажаним. Інша справа, коли класи пов'язані логікою програми:

```
class Student; // неповне оголошення класу
class Teacher{
   friend void registration();
  protected:
   int Nstudents;
   Student *plist[100];
   /// і т. д.
};
class Student{
   friend void registration();
  protected:
   Teacher *Tch;
   int semesterhours;
   /// і т. д.
};
```

У вищенаведеному прикладі функція *registration()* може застосовуватися як класом *Teacher*, так і класом *Student*, не знаходячись у жодному з них, але пов'язуючи їх під час реєстрації (ініціалізації), що цілком нормально, виходячи з даного контексту.

Тепер щодо випадку, коли використовуються функції-друзі як елементи-функції. Зазвичай, дружньою у класі оголошується елемент-функція іншого класу. Ця функція матиме вільний доступ до захищеної та закритої частин класу, в якому вона оголошена як *friend*. На практиці це може виглядати таким чином:

```
class A; // неповне оголошення класу
class B
{
   private:
    char *s2;
    public:
    B(){ s2= "B!!! ";}
    void show(A &c1);
};
class A
{
    friend void B::show(A&c1);
   private:
    char *s1;
    public:
    A() {s1="A !!!";}
};
```

```
void main()
{
   A c1;
   B c2;
   c2.show(c1);
}
void B::show(A&c1)
{
   cout<< c1.s1<<s2<<endl;
}
```

В реалізації видно, як функція *show()* звертається до закритої частини обох класів, хоча належить лише одному з них - класу *B*. Слід відмітити, що клас, в якому міститься прототип елемента-функції, повинен оголошуватися раніше класу, що вказує на функцію-елемент як дружню. Так для класу *A*, який оголошує дружньою для себе функцію *B::show(A&c1),* оголошення класу *B* попередньо повинно бути доступним компілятору.

Дружність у будь-якому випадку є порушенням цілісності даних, тому навряд чи є сенс у тому, аби надмірно зловживати нею. Деяке конкретне використання *friend*-функції знаходять саме при перевантаженні операторів при маніпуляції екземплярами класів, про що йтиметься в наступних розділах.

2.7 Поліморфізм

2.7.1 Віртуальні функції

Нагадаємо, що покажчику на базовий клас можна присвоїти значення адреси об'єкту будь-якого похідного класу (див. розділ 2.6.1."Механізм успадкування"). При цьому виклик методів через такий покажчик відбувається у відповідності до типу покажчика, а не до фактичного типу об'єкта, на який він посилається в конкретний момент. Продемонструємо це на прикладі, для чого знову ж таки повернемося до вже згадуваних класів у попередніх розділах про службовців та менеджерів - *employee* та *manager*, та додамо до їх протоколів однойменні функції, що друкують власну класову інформацію:

```
class employee {
     char* name;
   public:
     employee* next;
     void print()
     { cout <<"employee …" ; };
     // ...
};
class manager : public employee {
   public:
     void print(){
        cout<< " manager …";};
        // ...
     };
```

Тут слід відповісти на декілька запитань, що стосуються виклику однойменних функцій у головній програмі. Яким чином відбудеться виклик однойменної функції при активації об'єкта похідного класу?

```
void main()
{
   employee e, *eptr;
   manager m, *mptr;
   e.print();      // клас employee …
   m.print();      // клас manager …
   mptr=&m;
   mptr->print();  // клас manager …
   eptr=mptr;      // маємо право!
   eptr->print();// а тут ні - знову клас employee (!?)
}
```

Все нібито закономірно та не викликає сумнівів, окрім останнього оператора. Коли ми звертаємося до функції похідного об'єкту, використовуючи покажчик на базовий клас, викликається функція базового класу! Цей процес носить назву раннього зв'язку "клас+метод", коли зв'язки з методами встановлюються жорстко на етапі компоновки програми. Щоб викликати метод класу *manager*, слід застосувати явне перетворення типу покажчика:

```
((manager*)eptr)->print();
// після перетворення - клас manager
```

Але усувати такі "непорозуміння" можна й іншим, більш гнучким шляхом – оголосити *print() віртуальною* функцією. Вона відрізнятиметься від звичайної функції-елемента лише додаванням ключового слова *virtual:*

```
virtual void print ();
```

У спрощеному розумінні *віртуальна* функція – це функція, виклик якої залежить від типу об'єкта. В традиційному розумінні ми спочатку "прив'язали" об'єкт даних до функції, тобто раніше зв'язок *"об'єкт + метод"* визначався б на етапі написання коду. Оголосивши функцію віртуальною, ми підключаємо механізм пізнього зв'язку, коли визначення конкретного посилання на метод відбуватиметься на етапі виконання програми в залежності від типу об'єкта, який викликав метод. Оскільки ми ведемо мову про об'єктно-орієнтоване програмування, у нас з'являється набагато ефективніша можливість писати віртуальні функції, щоб сам об'єкт міг визначити, яку саме функцію необхідно активізувати під час виконання програми.

Але перш, ніж дійсно зрозуміти віртуальність, слід більш детальніше зупинитися на одному з найважливіших принципів класів, пов'язаних відношенням успадкування. Згідно об'єктно-орієнтованої парадигми, покажчик на базовий клас може посилатися не лише на об'єкт свого класу, але й на об'єкт іншого класу, похідного від базового (про це вже згадувалося у попередньому розділі: оскільки менеджер є службовцем, покажчик на *employee* може посилатися не тільки на об'єкт свого класу, але й на похідний об'єкт цього класу *manager*). Цей принцип стає особливо важливим, коли в класах, пов'язаних відношенням успадкування, визначаються віртуальні функції. Знову повернемося до вже відомих нам класових протоколів, але вже з віртуальними функціями:

```
class employee {
   public:
    virtual void print() {
       cout <<"employee …" ;};
};
class manager : public employee {
   public:
    virtual void print(){
      cout<< " manager …";};
};
void main() {
   employee *eptr;
   eptr=new manager;
   eptr->print();
}
```

Ми оголосили у головній програмі покажчик на базовий клас **eptr* та присвоїли йому адресу новоствореного об'єкту похідного класу у динамічній пам'яті (покажчик *eptr* може зберігати адресу об'єкту не лише типу *employee*, але й *manager*). Викличемо віртуальну функцію

eptr->print(). Ніякого приведення типу вже не потрібно: гарантовано, що під час виконання програми цей оператор викличе підходящу віртуальну функцію того класу, на об'єкт якого в даний момент посилається *eptr,* а саме *manager::print()*.

Деякі моменти опису та використання віртуальних функцій можна перерахувати у такому порядку:

1. Якщо метод, визначений у базовому класі, як віртуальний, також визначається у похідному класі з тим же ім’ям та списком параметрів, тоді він автоматично є також віртуальним. Віртуальні методи успадковуються, тобто перевизначення їх у похідному класі необхідно лише тоді, коли необхідно задати відмінні дії при виконанні цього методу у даному класі. При цьому права доступу при перевизначенні змінити неможливо.
2. Зарезервоване ключове слово *virtual* вказує компілятору на побудову таблиці віртуальних правил *VMT* (*virtual method table*), що міститиме адреси таких функцій для даного класу. Кожний представник класу з віртуальною функцією містить покажчик *VPTR* (*virtual pointer*) на його таблицю віртуальних методів *VMT*.
3. На етапі компіляції у початок конструктора автоматично вставляється покажчик *VPTR* на таблицю віртуальних правил *VMT* .
4. Адреса деякої віртуальної функції має одне й те ж саме зміщення в таблицях *VMT* кожного класу конкретної ієрархії.
5. При активації віртуальних методів згенерований код спочатку знаходить покажчик *VPTR* на таблицю *VMT*, а потім з таблиці вибирає адресу віртуальної функції, та, насамкінець, проводить безпосередній виклик функції.
6. Віртуальний механізм працює лише за допомогою покажчиків (посилань) на об'єкти. Об’єкт, що містить віртуальні функції, та визначений через покажчик або посилання, носить назву поліморфного. У даному випадку поліморфізм полягає у тому, що за допомогою одного й того ж звертання до методу виконуються різні дії в залежності від типу, на який посилається покажчик у даний момент часу.
7. Віртуальна функція не може бути оголошеною як *static*, але може бути оголошена, як дружня.

8. У базових класах рекомендується використовувати віртуальний деструктор.

Отже, виклики віртуальних функцій-членів визначаються під час виконання програми (що носить назву пізнього або динамічного зв'язування) на відміну від звичайних елементів-функцій, коли зв'язок "об'єкт + метод" визначається на етапі компіляції (як раннє або статичне зв'язування). Термін "пізнє зв'язування" іноді замінюють терміном поліморфізм (від грецької - різноманітний).

В чому полягає практичне значення поліморфізму в реальному програмуванні? Чіткого правила, за яким слід оголошувати методи віртуальним немає. Можна тільки дати рекомендацію оголошувати віртуальними методи, якщо існує вірогідність їх перевизначення у похідних класах. З іншого боку, при проектуванні ієрархії не завжди можна передбачити, яким чином будуть розширюватися базові класи у майбутньому, особливо при проектуванні обєктно-орієнтованих бібліотек класів. Тут поліморфізм дійсно дуже важливий, оскільки механізм успадкування без нього майже не принесе користі. (До речі, мови, в яких поліморфізм не підтримується, наприклад Ада, взагалі носять назви об'єктних.).

Виклик віртуальної функції реалізується як непрямий виклик за таблицею віртуальних правил. Ця таблиця створюється компілятором під час компіляції, а зв'язок відбувається під час виконання.

Не існує "ідеальної" ієрархії класів для кожної конкретної програми. По мірі просування на шляху розробки може виявитися ситуація, що доведеться вводити нові класи, які докорінно змінять усю ієрархію. Нічого дивного тут немає, адже кожна ієрархія класів являє собою поєднання експериментальних досліджень та інтуїції, основаних на практиці.

2.7.2 Чисті віртуальні функції та абстрактні базові класи

У ситуаціях, коли віртуальні функції викликаються у похідному класі, але не визначаються у ньому, викликається функція базового класу. Однак є чимало випадків, коли просто не існує смислового визначення віртуальної функції у базовому класі.

Так, уявімо собі, що ми пишемо новий проект для зображення різних геометричних фігур на екрані і прагнемо спроектувати класи для таких спеціалізованих плоских фігур, таких як трикутник, прямокутник, коло тощо. Оскільки усі ці класи тісно взаємопов'язані,

спочатку є сенс створити деякий базовий клас узагальненої фігури - *Shape*, що міститиме загальні атрибути будь-якої фігури.

Розглянемо ситуацію, коли однією з обов'язкових функцій в ієрархії буде віртуальна функція *Area*(), що друкує значення площі, яку займає фігура. Зрозуміло, що ми не можемо однозначно сказати, яке смислове значення цієї функції буде у базовому класі *Spape*, оскільки не можна визначити площу узагальненої фігури. З другого боку, кожний похідний клас в змозі визначити для себе власну версію цієї функції, що відповідає принципам поліморфізму в даній ситуації. Як тоді бути з визначенням такої функції у базовому класі?

Розв'язанням цієї проблеми є оголошення функції *Area*() як чистої *віртуальної функції* у базовому класі.

Чиста віртуальна функція (*pure virtual function*)- це віртуальна функція, що не має визначення у базовому класі даної ієрархії. У такому випадку її тіло визначається, як чистий специфікатор, наприклад, як у нашому випадку :

```
virtual void Area()=0;
// чиста віртуальна функція Area()
```

Такий опис нагадує звичайний прототип, що умовно прирівнюється нулю, який є ознакою чистої віртуальної функції. Якщо клас містить принаймні одну чисту віртуальну функцію, його часто іменують *абстрактним класом* (*abstract class*). Його особливість така, що не можна створити екземпляр такого об'єкта, а лише покажчик або посилання на цей клас, тому що він містить невизначену віртуальну функцію. Загальна ідея використання абстрактних класів полягає у тому, щоб по мірі просування вниз по ієрархії класів останні набували більшої спеціалізації та ставали "менш абстрактними". Зрозуміло, що класи, які знаходяться у кінці лінії успадкування, повинні визначати усі абстрактні функції.

Отже, при оголошенні чистої віртуальної функції її поведінка не визначається: лише кожний похідний клас, що не є абстрактним, повинен визначати, що робитиме дана функція. Таким чином, зручним є використання абстрактного базового класу до проектування ієрархії у задачі, згаданої вище:

```
#include<iostream.h>
class Shape
{
  protected:
   double x,y; // змінні-координати точки
   int col;
```

```
   //...
  public:
   void set( double ix, double iy=0)
   {
     x=ix;
     y=iy;
   }
   virtual void area()=0; // чиста віртуальна функція
Area()
};
class Triangle: public Shape
{
  public:
  void area()
  {
    cout<<"Трикутник з висотою "<<x<<" та основою "<<y;
    cout<<" має площу "<<0.5*x*y<<"\n";
  }
};

class Rectangle: public Shape
{
 public:
 void area()
 {
   cout<<"Прямокутник зі сторонами "<< x <<" та "<<y;
   cout<<" має площу "<<x*y<<"\n";
 }
};

class Circle: public Shape
{
  public:
  void area()
  {
    cout<<"Коло з радіусом "<< x <<" має площу"
    <<3.14159*x*x<<"\n";
  }
};

main()
{
  Shape *p; // оголошення на базовий тип
            // (створити об'єкт не можна!
  Triangle T;
  Rectangle R;
  Circle C;
  p=&T;
```

```
  p->set(2,7);
  p->area();
  p=&R;
  p->set(4,6);
  p->area();
  p=&C;
  p->set(7);
  p->area();
}
```

Функція *area*(), яку можна обрахувати лише виходячи з опису самої фігури, визначається в усіх похідних класах з набуттям спеціалізації. Якщо цього не зробити у даній ієрархії, компілятор видасть повідомлення про помилку. Якщо доопрацювати цей проект у більш практичний бік, можна було б додати ще декілька віртуальних функцій, що також могли б бути оголошені, як чисті віртуальні:

```
virtual void draw();  // виведення фігури
virtual void move(int regime) {….}
                 // переміщення фігури згідно режиму
virtual void rotate(int i) {….}
                // обертання фігури на заданий кут
```

Якщо у похідному класі, успадкованому від абстрактного базового, не відбулося заміщення чистої віртуальної функції, він також автоматично стане абстрактним. При подальшому успадкуванні рано чи пізно виникне потреба обов'язково визначити чисту віртуальну функцію саме у тому класі, який безпосередньо буде використовуватися для створення спеціалізованого об'єкту.

2.7.3 Розміщення VPTR та таблиці VMT у пам'яті

Розглянемо більш детально, яким саме чином під час виконання програми відбувається пошук підходящої віртуальної функції, що був описаний у попередньому розділі. Для цього використаємо деяку спрощену ієрархію класів, яка містить одну віртуальну функцію:

```
#include<stdio.h>
class A {
  int a;
 public:
  void f()
  {
     puts("From A::f()");
  }
```

```
  virtual void g()
  {
     puts("From A::g()");
  }
};
class B: public A
{
  int b;
 public:
  void f()
  {
     puts("From B::f()");
  }
  virtual void g()
  {
    puts("From B::g()");
  }
};
void Do_Something(A& a)
{
   a.f();
   a.g();
}
void main()
{
   A a;
   B b;
   Do_Something (a);
   Do_Something (b);
}
```

Перш за все відмітимо, що ієрархія класів використовує одночасно віртуальні та звичайні функції. Результат роботи такої програми виглядає так:

```
From A::f()
From A::g()
From A::f()
From B::g()
```

Як видно, звертання до функції *Do_Something(A&)* з об'єктом класу B приведе до виклику функцій *A::f()* та *B::g()*. Це відбувається тому, що у виклику функції *Do_Something (b)* аргумент посилання *B&* неявно перетворюється на *A&*. Адже видно, що *A::f()* - за визначенням невіртуальна функція, тому було згенеровано код прямого звертання до функції базового класу. Інша справа, коли викликається *a.g()* в *Do_Something(A&):* це призводить до генерації коду, який включає

механізм обробки віртуальних функцій. Схематично виділення пам'яті під розміщення об'єктів може бути зображено таким чином (рис 2.4):

Рис. 2.4. Схема розміщення об'єктів у пам'яті

Під час виконання код використовує таблицю віртуальних правил *VMT*. За покажчиком *vptr* відбувається визначення, яку саме функцію слід викликати при звертанні до *g()*. Слід відмітити, що поле *vptr* класу B має одне і той самий зсув, що і *vptr* у класі A, отже, *Do_Something(A&)* отримує доступ до правильної функції, незалежно від того, який саме передається в *Do_Something(A&)*. Єдина та ключова відмінність полягає якраз у тому, що *vptr* для об'єктів класу A та *vptr* для об'єктів класу B вказують на власні, а отже, різні таблиці віртуальних правил *VMT*. Таким чином і реалізується механізм пізнього зв'язування.

2.7.4 Віртуальні деструктори

Явний опис деструкторів у програмах потрібний лише тоді, коли об'єкт створюється у динамічній пам'яті. При використанні віртуальних деструкторів досить очевидними є переваги поліморфізму. Зазвичай, вони застосовуються тоді, коли при знищенні об'єктів необхідно видалити об'єкти похідного класу, на які посилаються покажчики на базовий клас. Саме це демонструє нижченаведений приклад.

```
#include<string.h>
class TBase
{
   private:
     char *sp1;
   public:
     TBase(const  char *s)
     {
        sp1=strdup(s);
     }
     virtual ~TBase()
     {
        delete sp1;
     }
};
class TDerived: public TBase
{
   private:
     char *sp2;
   public:
     TDerived(const char*s1, const char *s2):TBase (s1)
     {
        sp2=strdup(s2);
     }
     virtual ~TDerived()
     {
        delete sp2;
     }
};
```

У головній програмі організуємо демонстрацію створення та знищення об'єкту похідного класу *TDerived*:

```
void main()
{
  TBase *pbase;
  pbase=new TDerived("String 1","String 2");
  delete pbase;
}
```

Проведемо деяке додаткове спостереження даного коду та зробимо деструктор на деякий час звичайним, невіртуальним. Тоді останнім оператором головної програми при видаленні об'єкта через покажчик базового класу було б викликано лише деструктор базового класу, залишивши у динамічній пам'яті незвільнений рядок у вигляді *sp2*. Ця проблема розв'язується оголошенням віртуальних деструкторів: гарантовано, що у разі, коли знищується об'єкт класу *TDerived*, адресований покажчиком на *TBase*, буде знищено обидва рядки.

Якщо при оголошенні деструктору базового класу його було оголошено, як віртуальний, усі деструктори похідних класів також будуть віртуальними. На відміну від динамічних об'єктів в ієрархії, не має сенсу турбуватися про віртуальність деструкторів, що створюються у статичній пам'яті. Як правило, у таких випадках буде організовано правильний порядок виклику деструкторів у ієрархії, згенерованих компілятором по замовчуванню.

2.8 Перевантаження операцій

Ключове слово *operator* використовується для того, аби визначити нову, перевантажену дію конкретного оператора мови. Як і у випадку з перевантаженими функціями, компілятор відрізнятиме різні функції за контекстом звертання числом і типом параметрів та операндів. Перевантаження операторів не вносить нічого нового у мові, з чим би ми не зустрічалися раніше, якщо лише згадати перевантаження функцій (див. розділ 2.2.5 "Перевантаження функцій"). Це дає можливість використовувати об’єкти у виразах замість того, щоб передавати їх як параметри у функції. Кожному оператору мова Сі++ ставить у відповідність ім’я функції, що складається з ключового слова *operator*, власне оператору та аргументів відповідних типів:

<тип повертання> operator<символ оператору> (<вхідні параметри>);

Приміром, оголошення функції *operator+* , що приймає два аргументи типу T та повертає значення у вигляді сумування двох значень матиме наступний вигляд:

```
T operator+(T a, T a);
```

Семантика такого запису повністю аналогічна функції *T add (T a, T b)*, яка, можливо, і більш звична для сприйняття, проте спосіб перевантаження оператора має певну вигоду. Як перевантажений оператор, функція *operator+()* може викликатися у виразах скрізь, де використовується знак + та мають місце представники класу *T*:

```
T a1, a2;
a1+a2;  // аналогічно operator + (a1,a2); або
add(a1,a2);
```

Перевантаження операцій підпорядковується наступним правилам:

- при перевантаженні зберігаються кількість аргументів, пріоритети операцій та правила асоціації, що використовуються у стандартних типах даних;
- для стандартних типів даних операції не підлягають перевизначенню;
- перевантажена функція-оператор не може мати параметрів по замовчуванню, не успадковується та не може бути визначеною як *static*.
- функція-оператор може бути визначена трьома способами - метод класу, дружня функція або звичайна функція. В останніх двох випадках вона повинна приймати хоча б один аргумент, що має тип класу, покажчика або посилання на клас.

Як перший приклад, що використовує перевантажені зовнішні *friend*-функції, розглянемо перевантаження бінарних операцій "+" та "-":

Приклад 1.

```
// клас приймає рядок як константу, перетворюючи його в
// еквівалент типу long
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class T
{
   private:
     char value[20];
   public:
     T(){value[0]=0;}
     T(const char *s);
     long getvalue(void)
     {
        return atol(value);
     }
     // функції оголошуються як друзі, отже мають
     // доступ до усіх  членів об'єкту T
     friend long operator +(T a, T b);
     friend long operator -(T a, T b);
};
T::T(const char *s)
{
    strncpy(value,s,11);
    value[11]=0;
}
```

```
long operator+( T a, T b)
{
    return (atol(a.value)+atol(b.value));
}
long operator-( T a, T b)
{
    return (atol(a.value)-atol(b.value));
}
main() {
   T a="23";
     // ініціалізація об'єктів символьними рядками
   T b="22";
   cout<< "Value of a="<< a.getvalue();
   cout<< "\nValue of b="<< b.getvalue();
   cout<< "\n a+b=="<<(a+b);
       // використання перевантажених операторів
   cout<< "\n a-b=="<<(a-b)<<"\n";
   return 0;
}
```

Таким чином можливе вільне оперування значеннями за допомогою звичайних операторів (+ та –), не застосовуючи при цьому операції перетворення до рядків, якщо тільки вони будуть вищевказаного типу *T*. Перевантажені функції-оператори були зовнішніми, а тому обов'язково визначалися з ключовим словом *friend*, аби мати доступ до елементів-даних з протокольної частини класу.

Перевантажені функції-операції можуть бути і членами класу, причому обов'язково нестатичними (згадаємо, що такі функції мають прихований аргумент *this*). За рахунок цього і з'являється відмінність в реалізації такої функції перевантаження на відміну від подібної зовнішньої. Саме про це і йтиметься нижче. Розглянемо клас *Curr*, що використовується для представлення грошової форми національної валюти:

Приклад 2.

```
class Curr{
   protected:
    unsigned int grivnya; // значення у гривнях
    unsigned int cop; // значення у копійках
   public:
    Curr(unsigned int d, unsigned int c)
    {
       grivnya=d;
       cop=c;
       while(cop>=100){
         // приведення до коректного грошового вигляду
```

```
          grivnya++;
          cop-=100;
        }
    }
    Curr operator +( Curr& s2);
};
```

Звичайним є бажання застосувати операції додавання екземплярів такого класу, коли результатом буде також змінна типу *Curr*. Перевантаження оператору додавання суттєво покращить зовнішній значень у грошовому виразі:

```
// визначення класової функції operator +
Curr Curr::operator +( Curr& s2)
{
  //оператор + додає "поточний" екземпляр до s2, зберігаючи
  // результат в новій змінній
  cop=cop+s2.cop;
  grivnya=grivnya+s2.grivnya;
  Curr val (d,c);
  return val;
}
```

Для кращого порівняння наведемо паралельний виклад аналогічної зовнішньої *friend*-функції, що не є членом класу, який теж міг бути застосованим у даному випадку:

```
// додає s1 до s2, зберігаючи результат в новій змінній
friend Curr operator +(Curr& s1, Curr& s2)
{
   int c=s1.cop+s2.cop;
   int d=s1.grivnya+s2.grivnya;
   Curr val (d,c);
   return val;
}
```

Функції практично ідентичні, за винятком хіба що кількості параметрів. Оператор не змінює значення жодного із своїх аргументів, створюючи новий екземпляр та повертаючи його значення. Але якщо у зовнішній версії складаються екземпляри *s1* та *s2*, то у випадку функції-елемента класу першим виступатиме "поточний" екземпляр, для якого викликатиметься функція (саме той, на який і посилається покажчик *this*), а вже другим йтиме *s2*. На відміну від реалізації використання таких функцій для обох випадків не відрізнятиметься:

```
Curr sum1(5,50);
Curr sum2(7,55);
Curr sum(0,0);
Sum=sum1+sum2;
```

Отже, у функції-елементі, що реалізує перевантажений оператор, завжди на один аргумент менше, аніж в аналогічної зовнішньої функції, оскільки лівий аргумент у такому випадку завжди передається неявно. Ця відмінність явно проглядається і у перевантаженні унарних операцій:

- як дружніх функцій:

```
friend long operator-(Ta)
{
  return -atol(a.value);
}
```

- як функцій-елементів:

```
long T::operator -(void)
{
  return -atol(value);
} // або -atol(this->value);
```

Зауваження щодо перевантаження операцій.

1. Існують обмеження на перевантаження: не підлягають цій процедурі селектор елемента структури (.), оператор доступу до елементу за покажчиком (*), операція дозволу видимості (::), символи препроцесору (*#, ##*) та *sizeof()*. Неможливим є введення власних операторів та зміна їх пріоритетів. Крім того, неможливо перевантажувати оператори для вбудованих типів.
2. Компілятор C++ не розуміє семантики перевантаженого оператору, а отже, не нав'язує ніяких математичних концепцій. Ніщо вам не завадить перевантажити, скажімо, оператор інкременту в якості зменшення аргументу, проте навряд чи в цьому є хоч трохи здорового глузду. Якщо не слідувати традиційному осмисленню обчислень, результати роботи вашої програми для стороннього спостерігача можуть бути збентежуючими.
3. Не існує виведення складних операторів з простих: якщо ви перевантажили оператори *operator+* та *operator=*, це зовсім не

означає, що C++ обчислить вираз a+=b, оскільки ви не перевантажили *operator* +=.

4. Перевантаження бінарних операторів не тотожньо відносно перестановки аргументів місцями, тим більше, якщо вони різного типу. Оскільки, скажімо, operator*(double, T&) та operator*(T&, double) володіють різними параметрами, їх необхідно перевизначати окремо. Зручно це зробити, не створюючи новий код у другому операторі, а послатися у ньому на визначений перший оператор, змінивши лише порядок слідування аргументів:

```
T operator*(double f, T &s)
{
   //реалізація методу
}
```

```
T operator*(double f, T
&s)
{
   return f*s;
}
```

2.9 Шаблони

Подібно тому, як клас фактично являє собою схематичний опис побудови об'єктів, так і шаблон є схематичним описом побудови класів та функцій. Використовуючи шаблони, з'являється можливість створювати узагальнені специфікації для класів та функцій, що найчастіше носять назву параметризованих класів (*generic classes*) та параметризованих функцій (*generic functions*). Таким чином, за допомогою реалізації узагальнених функцій можна зменшити розмір та складність програми. Особливо корисними шаблони є саме в бібліотеках класів - тут вони вказують програмісту необхідні специфікації, приховуючи при цьому деталі справжньої реалізації.

2.9.1 Параметризовані функції

Шаблон функції декларується за допомогою ключового слова *template*. Це слово використовується для створення шаблону (каркасу), що в загальних рисах описує призначення функції та надає опис операцій – сутність алгоритму, що може застосовуватися до даних різних типів. При цьому конкретний тип даних, над яким функція повинна виконувати операції, передаватиметься їй на етапі компіляції. Загальна форма функції-шаблону має вигляд:

Синтаксис:

```
template <список_аргументів_шаблону>  тип
ім'я_функції(параметри)
{
   // тіло функції
}
```

Список аргументів шаблону складається з ключового слова *class* та ідентифікатору (-ів), що визначає його тип. Коли компілятор створюватиме конкретну версію цієї функції, він автоматично замінить цей параметр конкретним типом даних. Цей процес носить назву *інстанціювання шаблону*.

Зручним є створення прототипу шаблона функції у вигляді його попереднього оголошення. Таке оголошення інформує компілятор про наявність шаблону та його очікуваних параметрах, наприклад:

```
template <class T> void funk(T array[ ], sizearray);
```

Нижче наведений приклад використання параметризованої функції swap(T& x, T& y), що здійснює обмін значеннями між двома її параметрами:

Приклад 1.

```
#include "stdio.h"
template <class T> void swap(T& x, T& y);
int main()
{
   int a=5,b=10;
   float c=7,d=14;
   char ch='a',chh='A';
   swap(a,b); // виклик для цілих аргументів
   printf("a=%d, b=%d\n",a,b);
   swap <float>(c,d); // виклик для float аргументів
   printf("c=%f, d=%f\n",c,d);
   swap <char>(ch,chh); // виклик для char аргументів
   printf("ch=%c, chh=%c\n",ch,chh);
   return 0;
}

// визначення параметризованої функції
template <class T> void swap(T& x, T& y)
{
    T temp=x;
    x=y;
    y=temp;
}
```

Програма відкривається прототипом шаблонної функції. Аналізуючи приклад, зауважимо, що для інстанціювання шаблону застосовуються дві форми.

Конкретний тип для цього визначається компілятором автоматично, виходячи з типів параметрів у місці виклику функції (у наведеному вище прикладі це тип *int*), або задається явним способом (у прикладі це підстановка типів *float* та *char*).

Якби *swap()* була звичайною функцією, то потрібною була б її реалізація. Оскільки це шаблонна функція, компілятор сам реалізуватиме код такої функції, замінивши у даному випадку тип *Swap* на *int*, *float* та *char*. Таким чином можна зменшити розмір та складність програми, запропонувавши компілятору реалізацію узагальнених функцій.

Приклад 2. Функція приймає масив невизначеного типу та два цілих, обмінюючи зміст елементів масиву з індексами x та y

```
template <class T> void change (T t[ ], int x, int y)
{
   T tmp=t[x];
   t[x]=t[y];
   t[y]=tmp;
}
```

Коректними будуть такі виклики функцій за цим шаблоном:

```
double ar[7]={2,3,4,5,6,7,8};
change(ar,2,3); // або change <double>(ar,2,3);
char car[5]={'a','b','c','d','e'};
change(car,1,4); // або change <char>(car,1,4);
```

Таким чином, є можливість легко створювати нові функції, підставляючи в існуючі шаблони конкретні аргументи.

2.9.2 Параметризовані класи.

Визначаючи параметризований клас, ми створюємо його каркас (шаблон), що описує усі алгоритми, які використовуються класом. Фактичний тип даних, над яким проводитимуться маніпуляції, буде вказаний в якості параметру при конкретизації об’єктів цього класу. Компілятор автоматично згенерує відповідний об’єкт на основі вказаного типу. Загальна форма декларування параметризованого класу буде такою:

```
template <class Type> class class_name
{
   // протокольна частина класу
}
```

Визначення тіла класу аналогічно звичайному визначенню плюс використанню списку аргументів шаблону. Тип *Type* являє собою ім'я типу шаблону, яке в кожному випадку реалізації буде замінюватися конкретним типом даних. Типи можуть бути як стандартні, так і визначені користувачем, для їх опису завжди використовується ключове слово *class*. При необхідності можливо визначити більше одного параметризованого типу даних, використовуючи їх список через кому. У межах визначення класу шаблонне ім'я можна використовувати у будь-якому місці. Для створення конкретної реалізації використовується наступна форма:

```
class_name <type> ob;
```

У цьому випадку *type* представляє собою ім'я конкретного типу даних, над якими фактично оперуватиме клас, та замінює змінну *Type*. До речі, елементи-функції, над якими оперуватиме клас, автоматично стають параметризованими, тобто їх необов'язково декларувати за допомогою ключового слова *template* (див. розділ 2.9.1. "Параметризовані функції"). Якщо метод описується за межами шаблону, його заголовок повинен мати наступні елементи:

```
template <список_аргументів_шаблону> тип_функц

ім'я_класу <аргументи_шаблону>:: імя_функц (список_пар_функц)
{…}
```

При більш широкому розгляді шаблон класу - узагальнене визначення сімейства класів, яке може використовувати не лише довільні типи, а й константи. Синтаксис його узагальненого опису наступний:

```
template <список_аргументів_шаблону> class ім'я_класу
{
  // визначення класу
}
```

Аргумент (список_аргументів_шаблону) може складатися :

- з ключового слова *class* , за яким слідує ідентифікатор, який визначає параметризований тип (*типовий параметр*);
- з конкретного імені типу, за яким слідує ідентифікатор (*нетиповий параметр*, константа);

Відповідно до кожного з типів параметрів існує два правила використання шаблонів класу. Для створення представника шаблонного класу потрібно вказати ім'я шаблону зі списком аргументів, що заключений у кутові дужки в якості специфікатору типу. Список аргументів модифікується так:

- при використанні аргументів виду нетиповий параметр, тобто «*ім'я типу ідентифікатор*» заміна відбувається константним виразом.
- при використанні аргументів виду "типовий параметр", тобто "*class ідентифікатор*", список аргументів модифікується з іменем типу;

Створивши представника шаблонного класу, надалі можливо працювати з ним так само, як з представником звичайного класу.

Приклад 1. Створимо параметризовану чергу із застосуванням в якості аргументу шаблону типового параметру. Головна функція демонструє використання цілих черг та черг з плаваючою комою на основі створеного шаблону класу.

```
#include "iostream.h"
#include "stdlib.h"
template <class TypeQ> class queue
{
   TypeQ *q;
   int sloc,rloc;
   int length;
  public:
   queue(int size);
   ~queue()
   {
     delete [] q;
   }
   void qstore(TypeQ i);
           // розміщення елемента в кінець черги
   TypeQ qretrieve();
          // вилучення першого елемента з черги
};

template <class TypeQ> queue<TypeQ>::queue(int size)
{
   size++;
   q=new TypeQ[size];
```

```
   if (!q)
   {
      cout<<"Неможливо створити чергу!\n";
      exit(1);
   }
   length=size;
   sloc=rloc=0;
}

template <class TypeQ> void queue<TypeQ>::qstore(TypeQ
i)
{
   if (sloc+1==length)
   {
      cout<<"Черга переповнена!\n";
      return;
   }
   sloc++;
   q[sloc]=i;
}
template <class TypeQ> TypeQ queue<TypeQ>::qretrieve()
{
   if (rloc==sloc)
   {
      cout<<"Черга порожня!\n";
      return 0;
   }
   rloc++;
   return q[rloc]; }
   int main(){
   queue <int> a(5), b(5); //створення двох черг типу
int
   a.qstore(100);
   b.qstore(200);
   a.qstore(300);
   b.qstore(400);
   cout<< a.qretrieve()<<" ";
   cout<< a.qretrieve()<<" ";
   cout<< b.qretrieve()<<" ";
   cout<< b.qretrieve()<<endl;
   queue <float> f(5), e(5); //створення двох черг типу
float
   f.qstore(2.12);
   e.qstore(2.99);
   f.qstore(-30.00);
   e.qstore(1.986);
   cout<< f.qretrieve()<<" ";
   cout<< f.qretrieve()<<" ";
```

```
   cout<< e.qretrieve()<<" ";
   cout<< e.qretrieve()<<endl;
   return 0;
}
```

Кожна з черг міститься у динамічному масиві, що адресується покажчиком *q*. Розмір черги передається як параметр конструктору класу *queue,* що зберігається у члені клас*у length.*. Відповідно змінні *rloc* та *slo*c використовуються для індексації черги; перша вказує індекс елементу, що буде вилучений, друга містить адресу, за якою буде збережено наступний елемент. Тип *TypeQ* можна розглядати як деякий формальний параметр черги, на місце якого при компіляції буде підставлено конкретний тип даних.

Приклад 2. Створимо визначення класу із застосуванням в якості аргументу шаблону типового та нетипового параметру. У якості прикладу розглянемо клас, що містить блок пам'яті визначеного типу та довжини.

```
// параметр шаблону - тип та константа
template <class Type, int size>  class memo {
  private:
    Type *p;
  public:
    memo()
    {
       p=new Type[size];
    }
   ~memo()
    {
       delete [ ] p;
    }
   operator Type*();
};

template <class Type, int size>
 memo <Type, size> :: operator Type *()
{
   return p;
}
```

Створення представника такого класу можливо таким способом:
memo<float, 50> floatblock;

Константа тут виступає нетиповим параметром. Взагалі у такій якості можуть передаватися змінні цілого та типу перелічення, а також покажчики або посилання на об'єкт або функцію.

Найпоширеніше застосування шаблони класів знаходять при створенні контейнерних класів. Перевага тут у тому, що як тільки логіка, необхідна для підтримки контейнеру, визначена, він може бути застосований до будь-яких типів даних без необхідності будь-якої перебудови. Завдяки цьому одного разу написаний та відлагоджений контейнерний клас можна використовувати повторно. Добре відома бібліотека шаблонів класів фірми *Borland*, що включає контейнерні класи, які можуть включатися до програм для керування колекціями даних різних типів.

Декілька слів щодо перевантаження шаблонів класу та функцій. В той час, як є можливим перевантаження імен шаблонів функцій, неможливим є цей процес для імен шаблонів класів.

Наприклад, неможливо визначити одночасно *Tarray<class T> та Tarray<class T, int size)*, в той час як без проблем перевантажуються шаблони функцій. Адже ніщо не завадить описати декілька шаблонів функцій з одним і тим же ім'ям, якщо лише вони мають відмінне число або різний тип параметрів.

2.10 Класи потоків C++

Потік – абстрактне поняття, що відноситься до будь-якого перенесення даних. Читання даних з потоку називається вилученням, запис даних в потік називається поміщенням, або включенням. Потік визначається як послідовність байтів і не залежить від конкретного пристрою, з яким проходить обмін (оперативна пам'ять, файл на диску, клавіатура або принтер). Обмін даними з потоком для збільшення швидкості передачі даних здійснюється, як правило, через спеціальну область оперативної пам'яті – буфер.

За напрямом обміну потоки можна поділити на *вхідні* (дані вводяться в пам'ять), *вихідні* (дані виводяться з пам'яті) і *двонапрямлені* (ті, що допускають як вилучення, так і поміщення даних).

Технологія потоків значно відрізняється від звичайних засобів введення-виведення, які використовує Сі. Згадаймо відомі функції з традиційного *Сі - printf(), scanf()* та численних їхніх "родичів", які не передбачають ніякої перевірки типу, потребуючи від програміста чіткого дотриманні правил застосування аргументів, символів

форматування тощо. Компілятор в такому випадку не в змозі сигналізувати про невідповідність специфікацій формату прийнятим фактичним аргументам, а тому цілком покладає цю відповідальність на користувача, що на практиці нерідко закінчується помилкою виконання. Використання класів потоків настільки спрощене, що контроль за співпаданням кількості та типів аргументів перекладається на компілятор.

2.10.1 Визначені об'єкти-потоки

Механізм потоків C++ ґрунтується на *перевантаженні функцій* (*операцій*), що забезпечує для кожного типу даних, які передаються, виклик відповідної функції. Застосування процедур обмежено файловими потоками та деякими пристроями, доступ до яких можливий як до визначених потоків. Ці процедури не допускають розширення. Класи C++, завдяки *поліморфізму*, дозволяють одним і тим же процедурам працювати з потоками різних типів. Широке використання перевантажених функцій дозволяє бібліотеці потоків підтримувати однаковий інтерфейс *I/O*. Такий інтерфейс робить код більш розбірливим та сприяє кращому абстрагуванню даних. Крім того, застосування у *I/O* - класах перевантажених операцій приводить до більш простого та зрозумілого синтаксису.

Щоб забезпечити програмі доступ до бібліотеки потоків C++, необхідно включити заголовочний файл *iostream.h*; також можуть знадобитися файли *fstream.h* (файлове введення/виведення), *iomanip.h* (файл маніпуляторів) та *strstream.h* (резидентні потоки).

Бібліотека *iostream* має чотири визначених об'єкта потоку (таблиця 2.3). Всі вони асоційовані зі стандартним інтерфейсом I/O.

Таблиця 2.3. Потоки введеня-виведення

Ім'я	*Клас*	*Опис*
cin	istream	Асоціюється зі стандартним введенням (клавіатурою)
cout	ostream	Асоціюється зі стандартним виведенням (екраном)
cerr	ostream	Асоціюється зі стандартним пристроєм помилок (екраном) з небуферизованим виводом
clog	ostream	Асоціюється зі стандартним пристроєм помилок (екраном) з буферизованим виводом

2.10.2 Операції поміщення та вилучення

Бібліотека класів С++ передбачає два основних класи для введення та виведення: відповідно *istream* та *ostream*. Потокові оператори застосовуються наступним чином:

об'єкт_потоку_вводу **>>** змінна;

об'єкт_потоку_виводу **<<** змінна;

Примітка. Ці оператори не слід плутати з відомими побітовими операторами зсуву вліво та вправо. Образно кажучи, в розглядуваному контексті перевантажені оператори >> та << однозначно ясно вказують напрямок потоку даних від одного об'єкта до іншого.

Потік *cout (character out)* - стандартний символьний вихідний потік, що по замовчування відповідає пристрою виведення (дисплей). Клас *ostream* використовує для виведення перевантажену операцію лівого зсуву (**<<**). Якщо ця операція застосовується об’єктом-потоком, вона носить ім'я операції *поміщення* у потік (*insertion operation*). Наступний приклад друкує рядок, застосовуючи операцію *поміщення* до визначеного об’єкту *cout*:

```
#include<iostream.h>
int main(void)
{
    cout << "Hello!";
    return 0;
}
```

Проаналізуємо, як відпрацює такий фрагмент. По-перше, С++ визначить, що в даній операції << лівий аргумент *cout* має тип *ostream*, а правий – *char**. За цими даними у заголовочному файлі буде знайдений відповідний прототип функції - *ostream& operator<<(ostream&, char*)*. Після цього буде згенерований виклик функції поміщення у потік *operator<<* з рядком *‘Hello!’* та екземпляром *cout* в якості параметрів. Іншими словами, здійснюватиметься виклик стандартної бібліотечної функції *operator<<(cout, ‘Hello!’),* як того у даному випадку вимагає прототип.

Аналогом в традиційному Сі потоку *cout* відповідає функція *fprintf()* та ім'я пристрою виведення *stdout*. Розглянемо простий приклад, що застосовує у двох варіантах відповідно *printf()* (це тільки скорочена форма *fprintf(stdout, ...)*) та об'єкт *cout*:

1 варіант

```
double num=2.35;
printf ("show 1 : % ld \n", num);
// непередбачене "сміття" на виході !
```

2 варіант

```
double num=2.35;
cout<<"show 2="<<num<<"\n";
// а тут все буде правильно
```

Саме тут і впадає в очі перевага потоків. Символи *% ld* в першому варіанті повідомляють про те, що змінна *num* відноситься до типу *long*. У наступному рядку при активації функції виявляється невідповідність типів, і *printf()* мовчки виконує неправильне виведення. Застосовуючи у другому варіанті об'єкт-потік, подібна помилка просто виключена - інформація про тип автоматично надходить безпосередньо від самого об'єкту *cout*.

Потік *cin (character in)* - стандартний символьний потік введення, що по замовчування відповідає клавіатурі. Клас *istream* використовує для введення перевантажену операцію правого зсуву (>>). У розглядуваному контексті її називають операцією вилучення з потоку (*extraction operation*). Аналогом цього потоку в Сі відповідає функція *fscanf()* та ім'я пристрою *stdin*. Наступний приклад застосовує операцію вилучення зі стандартного потоку для визначеного об’єкту *cin*, аби прочитати рядок з клавіатури:

```
#include<iostream.h>
int main(void)
{
   char name [100];
   cout << " Введіть ваше ім’я: ";
   cin >> name;
   cout << " Привіт, ";
   cout << name;
   return 0;
}
```

Класи *istream* та *ostream* перевантажують відповідно >> та << для усіх вбудованих типів даних. Таке перевантаження дозволяє використовувати однаковий синтаксис для роботи зі змінними різних типів. Наступний приклад ілюструє тотожність синтаксису в такій ситуації:

```
#include <iostream.h>

int main (void)
{
   char c = 'A';
   signed char sc = 'B';
   unsigned char uc = 'C';
   int i = 0xd;
   float f = 1.7;
   double d = 2.8;
   cout << c;   // Викликає operator << (char)
   cout << sc;  // Викликає operator << (signed char)
   cout << uc;  // Викликає operator << (unsigned char)
   cout << i;   // Викликає operator << (int)
   cout << f;   // Викликає operator << (float)
   cout << d;   // Викликає operator << (double)
   return 0;
}
```

2.10.3 Переадресація введення та виведення

Можна перевизначити імена *cin* або *cout* власним об'єктам-потокам. Таке призначення дозволяє програмі легко переадресувати стандартні операції введення або виведення. Наприклад, як у такому фрагменті:

```
#include <iostream.h>
#include <fstream.h>
const int MAX_LINE = 80;
ifstream ifs; // Переадресований вхідний поток
int main (int argc, char *argv [] )
{
   if ( argc>1 ) // Якщо вказаний аргумент ...
   {
      ifs.open (argv [1] ); // Спроба відкрити файл
      // Якщо успішна, переадресувати введення
      if ( ifs )  cin = ifs;
   }
   cout << " Введіть рядок тексту: ";
   // Прочитати дані зі стандартного вводу
   char line [MAX_LINE];
   cin.getline ( line, sizeof (line));
   cout << endl << " Ви ввели: " << line;
                        // Показати введені  дані ...
   return 0;
}
```

Похідні класи *ofstream*, *ifstream* та *fxtream* визначаються у заголовочному файлі *fstream.h* та застосовуються для операцій роботи з файлами, так само, як функції, що виконують аналогічну роботу в Сі (*fprintf(), fscanf(), fopen(), fclose()* тощо). Ці класи спільно використовують ряд функцій-елементів для керуванням процесом вводу-виведення, чимало з яких успадковуються саме від класів *istream* та *ostream.*

2.10.4 Визначення потокових операцій як дружніх

При використанні класів, як правило, прийнято оголошувати операції вилучення та поміщення друзями вашого класу. Таке оголошення забезпечує операції доступу до окремих елементів даних при форматуванні процесів I/O.

```
#include <iostream.h>
class TPiece
{
  // ...( Окремі дані )
 public:
  // ...
  friend istream& operator >> (istream&, TPiece&);
  friend ostream& operator << (ostream, const TPiece&);
};
// Підтримка вихідного потоку
ostream& operator << (ostream &os, const TPiece &p)
{
   // Може використовувати для форматування окремі дані
   return os;
}
// і т.д.
```

2.10.5 Функції керування процесом I/O

Для читання та встановлення ширини поля потоку у класі *ios* є функція *width* (таблиця 2.4).

Таблиця 2.4. Функції читання та встановлення ширини поля потоку

Функція	*Опис*
int ios::width ();	Повертає поточне значення внутрішньої змінної ширини поля потоку.
int ios::width (int);	Встановлює значення внутрішньої змінної ширини поля.

Для читання або зміни поточного заповнюючого символу можна застосовувати функції *ios::fill* (таблиця 2.5).

Таблиця 2.5. Метод fill класу ios

Функція	*Опис*
char ios::fill ();	Повертає поточний символ заповнення
char ios::fill (char);	Встановлює внутрішній заповнюючий символ потоку та повертає його попереднє значення

Функції *ios::precision()* можуть застосовуватися при виведенні чисел з плаваючою крапкою, дозволяючи читати або встановлювати поточне число значущих цифр (таблиця 2.6).

Таблиця 2.6. Метод precision класу ios

Функція	*Опис*
int ios::precision (int);	Встановлює внутрішню змінну точності дійсних чисел потоку та повертає попереднє значення
int ios::precision ();	Повертає поточне значення точності

2.10.6 Прапорці форматування

У потоках C++ існують *прапорці формату* (таблиця 2.7). Вони вказують, яким чином форматується введення та виведення. Прапорці є *бітовими полями*, що зберігаються у змінній типу *long*.

Таблиця 2.7. Прапорці форматування

Прапорець	*Положення*	*Опис дії*
skipws	0x0001	Ігнорування пробільних символи при вилученні
left	0x0002	Вирівнювання за лівим краєм поля
right	0x0004	Вирівнювання за правим краєм поля
internal	0x0008	Знак числа виводиться за лівим краєм, число – за правим.
dec	0x0010	Десяткова система числення
oct	0x0020	Вісімкова система числення

hex	0x0040	Шістнадцяткова система числення
showbase	0x0080	Виводиться основа системи числення (0x для шістнадцяткових чисел і 0 для вісімкових)
showpoint	0x0100	При виведенні дійсних чисел друкувати десяткову крапку і дробову частину
uppercase	0x0200	При виведенні використовувати символи верхнього регістру
showpos	0x0400	Друкувати знак при виведенні додатних чисел
scientific	0x0800	Друкувати дійсні числа у формі мантиси з порядком
fixed	0x1000	Друкувати числа у формі із фіксованою точкою
unitbuf	0x2000	Вивантажувати буфери всіх потоків після кожного виведення
stdio	0x4000	Вивантажувати буфери потоків stdout і stderr після кожного виведення

Розглянемо приклад із застосування прапорців формату iostream:

```
#include <iostream.h>
int main (void)
{
   int x = 1678;
   // Показати значення
   cout << "Значення x = " << x << '\n';
   // Зберегти значення прапорців
   long savedFlags = cout.flags;
   // Встановити основу 16 з індикацією
   cout.setf (ios::showbase | ios::hex);
   // Вивести значення знову
   cout << "Значення x = " << x << '\n';
   return 0;
}
```

2.10.7 Маніпулятори

Маніпулятори є функціями (знаходяться в *iomanip.h*), які можна включати у низку послідовних операцій поміщення та вилучення. Це зручний спосіб керування прапорцями потоку. Однак застосування

маніпуляторів не обмежується модифікаціями формату *I/O*. За виключенням *setw*, зміни, внесені маніпуляторами, зберігаються до наступної модифікації.

Маніпулятори, що не потребують застосування аргументів, називаються простими (таблиця 2.8).

Таблиця 2.8. Прості маніпулятори

Функція	*Опис, дія маніпулятору*
endl	Поміщує у вихідний потік символ нового рядка (\n) та викликає маніпулятор flush
ends	Поміщує у вихідний потік нульовий символ (\0) завершення рядка
flush	Примусово записує усі вихідні дані на відповідні фізичні пристрої
dec	10-а система обчислення
hex	16-а система обчислення
oct	8-а система обчислення
ws	Ігнорування при вводі ведучих символів-пропусків

Наступний фрагмент використовує маніпулятор для виведення значення у шістнадцятирічній та десятковій формі:

```
unsigned v=12345;
cout<< "In hexadecimal v=="<<hex<<v<<'\n';
cout<< "In decimal v=="<<dec<<v<<'\n';
```

Аналогічно застосовуються маніпулятори й для операцій введення:

```
cout << "Enter the value in hex:";
cin>>hex>>v;
cout<<"Value in decimal == " << dec <<v;
```

Наступні маніпулятори використовуються лише з параметрами (таблиця 2.9):

Маніпулятори-функції потоків виведення повертають значення типу *ostream&*, іншими словами посилання на об'єкт *ostream*. Можна описати й власні маніпулятори, визначивши функцію такого посилального типу. Наприклад, маніпулятор дзвоника для потоку виведення може бути таким:

Таблиця 2.9. Маніпулятори з параметрами

Маніпулятор	*Опис, дія маніпулятору*
setw(int n)	Встановити ширину поля, що задана в параметрі
setbase(int n)	Встановити систему числення – 0,8,10, або 16
setfill(int c)	Використати символ заповнення при “вирівнюванні”
lock(ios &ir)	Заблокувати дескриптор файлу для посилання ir на потік I/O
unlock(ios &ir)	Розблокувати дескриптор файлу для посилання ir на потік I/O
setprecision(int n)	Встановити точність виведення значень з плаваючою комою
setiosflags(long f)	Встановити біти форматування, що вказані в f

```
ostream& bell (ostream&)
{
   return os << "\a"
}
```

Тоді використання в рядку виведення буде таким:

```
cout << bell<<"Ding !!!"
```

2.10.8 Файлові потоки

Стандартна бібліотека C++ містить три класи файлового введення/виведення :

ifsteram – клас вхідних файлових потоків

ofstream – клас вихідних файлових потоків

fstream – клас двоспрямованих файлових потоків

Кожний з перерахованих класів містить конструктори, які дають можливість створювати об’єкти цих класів. Розглянемо можливі форми виклику конструкторів :

Конструктори без параметрів створюють об’єкт відповідного класу без зв’язування його з файлом.

```
ifstream();
ofstream();
fstream();
```

Конструктори з параметрами створюють об'єкт відповідного класу, відкривають файл із вказаним ім'ям і зв'язують його з об'єктом.

```
ifstream(const char *name, int mode = ios::in);
ofstream(const char *name, int mode = ios::out | ios::trunc);
fstream(cont char *name, int mode = ios::in | ios::out);
```

Перший параметр в наведених конструкторах – ім'я файла, другий параметр – режим відкриття файла. Можна встановлювати такі режими відкриття файлів:

```
enum open_mode  {
  in    = 0x01, // відкрити для читання
  out   = 0x02, // відкрити для запису
  ate   = 0x04, // встановити покажчик на кінець файлу
  app   = 0x08, // відкрити для додавання в кінець файлу
  trunc  = 0x10,  // якщо файл існує, знищити
  nocreate = 0x20,// якщо файл не існує, видати помилку
  noreplace= 0x40, // якщо файл існує, видати помилку
  binary   = 0x80  // відкрити в бінарному режимі
};
```

Таблиця 2.10 ілюструє відповідність між бітовими масками класу *ios* та режимами відкриття файла засобами *stdio.h* :

Таблиця 2.10. Зв'язок між бітовими масками класу *ios* та режимами відкриття файла засобами stdio.h

<*stdio.h*>	*binary*	*in*	*out*	*trunc*	*app*
"w"			+		
"a"			+		+
"w"			+	+	
"r"		+			
"r+"		+	+		
"w+"		+	+	+	
"wb"	+		+		
"ab"	+		+		+
"wb"	+		+	+	
"rb"	+	+			
"r+b"	+	+	+		
"w+b"	+	+	+	+	

```
// Приклад виконує копіювання вмісту файла oldfile.txt
// у файл
// newfile.txt.
#include <fstream.h>
int main()
{
 char ch;
 ifstream f1 ("OLDFILE.TXT");
 ofstream f2 ("NEWFILE.TXT");
 if (!f1) cerr<<"Cannot open OLDFILE.TXT for input";
 if (!f2) cerr<<"Cannot open NEWFILE.TXT for output";
 while (f2 && f1.get(ch))
    f2.put(ch);
 return 0;
}
```

2.11 Контейнерні класи.

Класи, що містять у своєму протоколі один або декілька об'єктів або покажчиків на об'єкти, носять назву контейнерних класів (*class containers*). В даному випадку має місце відношення між класами типу "містить". Найчастіше таку побудову протоколів класів підказує логіка програми, адже не завжди лише за допомогою успадкування вдається адекватно відобразити всю сукупність складних взаємовідношень між класами.

В якості прикладу розглянемо наочний приклад з життя - протокол деякого класу *House*. Протокол цього класу містить поля - об'єкти класу *Room*, який у свою чергу - поля класу *Furniture*. Створивши такий проект, можна сказати, що будинок *House* має кімнати *Room*, в яких є меблі *Furniture*.

```
#include<iostream.h>
#include<string.h>
// протокол класу Furniture
class Furniture
{
  private:
    char woodtype[20];
    char furntype[25];
  public:
    Furniture(char* wood, char*sort)
    {
       cout <<"Constructor of furniture\n";
       strcpy(woodtype, wood);
       strcpy(sort, furntype);
    }
```

```
    friend ostream& operator<<(ostream& o, Furniture&
                                                  afurn);
};
ostream& operator<<(ostream& o, Furniture& afurn)
{
  cout<<"Furniture is"<<afurn.furntype<<"\n";
  cout<<"Furniture is made of "<<afurn.woodtype<<"\n";
}
// протокол класу Room
class Room
{
   private:
     Furniture furn1, furn2;
     unsigned length, width;
     char expouser[50];
     Room(Furniture p1, Furniture p2,
      unsigned l,unsigned w,  char* exp):
       furn1(p1), furn2(p2), length(l), width(w)
     {
        cout<<"Constructor of room\n";
        strcpy(expouser, exp);
     }
     friend ostream& operator<<(ostream& o,
                                          Room& aRoom);
};
ostream& operator<<(ostream& o, Room& aRoom)
{
   cout<<"Length is "<<aRoom.length<<"\n";
   cout<<"Width is "<<aRoom.width<<"\n";
   cout<<"Expose of room  --- "<<aRoom.expouser<<"\n";
   cout<<"Furniture 1 --- "<<aRoom.furn1<<"\n";
   cout<<"Furniture 2 --- "<<aRoom.furn2<<"\n";
}
// протокол класу House
class House
{
  private:
    Room r1, r2,r3;
    float price;
    unsigned square;
  public:
    House (Room ar1, Room ar2, Room ar3, float aprice,
       unsigned area): r1(ar1), r2(ar2), r3(ar3),
       price(aprice), square(area)
    {
       cout<< " Multiple constructor for house\n";
    }
```

```
    friend ostream& operator<<(ostream& o,
    House& aHouse);
};
ostream& operator<<(ostream& o, House& aHouse)
{
   cout<<"Price is "<<aHouse.price<<"\n";
   cout<<"Area is "<<aHouse.square<<"\n";
   cout<<"Room 1 --- "<<aHouse.r1<<"\n";
   cout<<"Room 2 --- "<<aHouse.r2<<"\n";
   cout<<"Room 3 --- "<<aHouse.r3<<"\n";
}
```

Якою буде послідовність виклику конструкторів при створенні об'єкту класу *House*? Спочатку буде проведено ініціалізацію усіх полів-об'єктів у протоколі класу у тому порядку, як вони оголошені. Тільки після цього ініціалізуватиметься сам клас, що містить ці поля. Оголошення змінної *House my_house(room1, room2, 5500.00, 50)* спричинить виклик довгого ланцюга конструкторів. Першими викликатимуться конструктори класу *Furniture*, після чого *Room*, і насамкінець, *House*.

Прикладами контейнерних класів можуть служити класи користувача, що описують масиви, лінійні списки або стеки. Характерно, що для кожного типу такого контейнеру можна визначити стандартні методи роботи з його елементами, які не залежать від конкретного типу даних, що зберігається у контейнері, тому один і той самий вид контейнеру можна використовувати для зберігання та обробки даних різних типів. Ця можливість реалізується за допомогою шаблонів класів, про що вже йшла вище мова у відповідному розділі. На сьогодні є добре відомою стандартна Сі++-бібліотека шаблонів STL(*Standart Template Library*), яка містить основні структури даних для написання програм, такі як вектори, черги, різновиди списків, множини та словники. На жаль, з причин обмеженого об'єму, її розгляд не входить до переліку тем даного посібника.

2.12 Вкладені класи.

Бувають ситуації, коли при оголошенні одного класу в середині його оголошення можуть бути оголошені інші класи. У таких випадках такі класи називають вкладеними класами (*nested classes*), а клас, що їх об'єднує - клас з об'єктною ідентичністю (*object identity*). Наведемо гіпотетичний приклад, що описує вкладеність класу *Nested* у класі C.

```
#include<iostream.h>
class C
{
    class Nested
    {
       int who;
      public:
       Nested(int a);
       ~Nested(void);
    };
   public:
    C(int b)
    {
       Nested (b*b);
       cout<<"Constructor C\n";
    }
    ~C(void)
    {
      cout<<"Destructor C\n";
    }
};
C::Nested::Nested(int a)
{
   who=a;
   cout<<"Constructor of Nested class \n";
}
C::Nested::~Nested(void)
{
   cout<<"Destructor of Nested class\n";
}
int main(int argc, char* argv[])
{
  C object(3);
  return 0;
}
```

В результаті роботи даної програми ми отримаємо:

```
Constructor of Nested class
Destructor of Nested class
Constructor C
Destructor C
```

Складно відразу визначити, в яких ситуаціях класи краще реалізувати якраз за рахунок вкладення. Так, даний підхід широко застосовується у широковідомій технології об'єктів *COM* (*Component Object Model*), основна з цілей якої полягає у відокремленні інтерфейсу класу від його реалізації.

2.13 Локальні класи.

Іноді клас може бути оголошений в середині функції. Такий клас носить назву локального класу (*local class*). Функція, в якій оголошений локальний клас, не має спеціального доступу до членів локального класу, а локальний клас не може мати статичних членів-даних.

Об'єкт локального класу може бути створений лише в середині функції, в області дії оголошення класу. Усі функції-члени локального класу повинні бути оголошеними в середині оголошеного класу, тобто бути *inline*-функціями по замовчуванню.

```
#include<iostream.h>
void f(void);
int main(void)
{
  // Буде викликано функцію, що містить локальний клас
  f();
  return 0;
}
void f(void)
{
   class Local
   {
      int who;
     public:
      Local(int a)
      {
         who=a;
         cout<<"Конструктор локального класу"
                                          <<who<<"\n";
      }
      ~Local(void)
      {
          cout<<"Деструктор локального класу"
                                          <<who<<"\n";
      }
   } loc_ob(1), loc_ob1(2);
}
```

В результаті роботи даної програми ми отримаємо:

```
Конструктор локального класу 1
Конструктор локального класу 2
Деструктор локального класу 2
Деструктор локального класу 1
```

Безпосередньо у тілі функції створюються два об'єкти локального класу *loc_ob* та *loc_ob1*. Після того, як функція завершує своє існування, обидва об'єкти автоматично знищуються, про що свідчить виклик деструкторів, визначених у протоколі оголошення класу. У якому випадку може бути корисним використання локальних класів? Наприклад, це можуть бути особливі ситуації, коли об'єкти мають обмежений час існування у програмі, або у тих випадках, коли вимагається гарантійне вивільнення пам'яті під великі масиви тимчасових об'єктів, які вже виконали покладену на них конкретну місію.

2.14 Обробка виняткових ситуацій

Виняткова ситуація у програмі - це виникнення деякої непередбачуваної або аварійної події, що потребує спеціальної обробки. Наприклад, це ділення на нуль або звертання за неіснуючою адресою пам'яті. Такі події, як правило, призводять до завершення програми з системним повідомленням про помилку. Сі++ дає можливість коректно обробляти такі ситуації, не припиняючи програми, продовжувати її виконання.

Слід зауважити, що обробка виняткових ситуацій в Сі++ не підтримує обробку асинхронних подій, таких, як помилки устаткування або обробку переривань. Цей механізм працює лише для подій, що виникають в результаті роботи самої програми, та можуть бути вказаними явно. У цьому випадку весь обчислювальний процес логічно розподіляється на дві частини - виявлення помилкової (аварійної) ситуації та її обробка.

Перейдемо до синтаксису виняткових ситуацій, для чого детально розглянемо оператори *try*, *throw* та *catch*. Усі три перераховані оператори носять назву операторів-обробників виняткових ситуацій в Сі++. Програма спроможна генерувати виняткові ситуації, якщо вона містить принаймні один *try-блок*, що генерує (збуджує) таку ситуацію. Такий блок завжди виокремлюється фігурними дужками:

```
try{
    … …. // тіло try-блока
}
```

Для генерації (збудження) виняткової ситуації виконується оператор *throw*, який може використовуватися з параметром, або без нього:

```
throw[вираз];
```

Оператор *throw* посилає об'єкт, що описує суть помилкової ситуації. Такий об'єкт може бути звичайним виразом певного типу - літеральним значенням, рядком, об'єктом класу тощо. Розглянемо деяку гіпотетичну функцію, в якій забезпечується підтримка збудження виняткових ситуацій двох видів:

```
int func()
{
   if (condition1) throw ("trouble 1");
   if (condition2) throw ClassA();
   return 0;
}
```

Обробка збуджених ситуацій починається з оператору *catch*, який перехоплює виняткову умову, згенеровану якимось процесом:

```
catch (тип)
{
    /* тіло обробника*/
}
```

Так, для обробки першої з двох помилкових ситуацій цей оператор може бути записаний так:

```
catch (char *message)
{
   cout<<"Error ---"<<message<<endl;
   exit(-1);
}
```

Тепер є все необхідне, щоб організувати обробку виняткових ситуацій вищенаведеної функції *int func()*, але перш за все її виклик слід обов'язково розмістити в середині блоку *try* :

```
try
{
   int a=func();
   cout<< "a=="<<a<<endl;
}
```

Таке розміщення виклику функції говорить про те, що вона не є просто звичайною функцією. Визначивши помилковий стан, функція спроможна збуджувати виняткову ситуацію, що викликає наступні наслідки:

- функція визначає, що дійсно виникла умова виняткової ситуації;
- функція спроможна збуджувати одну або декілька виняткових ситуацій різних типів, використовуючи оператор *throw,* що відповідають різним умовам виняткових ситуацій;

- як тільки оператор *throw* збуджує виняткову ситуацію, функція негайно завершує своє виконання;
- надалі функція "робить запит" на розв'язування проблеми обробником виняткової ситуації *catch*, що викликається автоматично на посилання об'єкта виняткової ситуації.

Для нашого прикладу за блоком *try* повинні слідувати декілька операторів-обробників *catch.* Тоді розглядуваний гіпотетичний приклад набуде закінченого вигляду:

```
class ClassA{
  // клас, що повідомляє про своє створення та знищення
  public:
   ClassA(){cout<<"Hello!"<<endl;}
   ~ClassA(){cout<<"Bye!"<<endl;}
};
int func()
{
   int a;
   a=….???; //  код помилкової ситуації
   if (a==0) throw ("Trouble 1");
   if (a==1) throw ClassA();
   return 0;
}
int main(void)
{
  try
  {
    cout<<"Входимо у try-блок"<<endl;
    int a=func();
    cout<<"Виходимо з try-блоку"<<endl;
    cout<< "a=="<<a<< endl;
  }


  catch (char *message)
  {
     cout<<"Error ---"<<message<<endl;
     exit(-1);
  }
  catch (ClassA)
  {
     cout<<" ClassA! "<<endl;
     exit(-2);
  }
}
```

Якщо функція збуджує виняткову ситуацію, виконання *try*-блоку негайно припиняється, і отримані об'єкти перехоплюються відповідними операторами *catch*, що обробляють дану ситуацію. У нашому випадку два оператори *catch* обробляють виняткові ситуації для рядкового значення та типу класу *ClassA*. Якщо при активації функції виконається перша умова, функція збуджує виняткову ситуацію з рядковим повідомленням, якщо виконується друга умова, функція посилає об'єкт типу *ClassA*, який створюється в даному випадку за допомогою звернення до конструктору класу по замовчуванню. У випадку, коли виняткових ситуацій не виникне, функція завершить свою роботу, повертаючи при цьому значення типу, з яким ця функція оголошена, в місце виклику цієї функції.

Виняткові ситуації відкидають необхідність резервування спеціальних значень у помилкових ситуаціях. Якщо у програмі виникли необроблені виняткові ситуації, то по замовчуванню викликається функція *unexpected()*. Найчастіше *unexpected()* викликає функцію *terminate()*, яка, в свою чергу, викликає функцію *abort()* для аварійного завершення програми. Є можливість замінити *unexpected()* та *terminate()* власними функціями-обробниками виняткових ситуацій.

ЧАСТИНА 3. ПЕРЕЛІК ЛАБОРАТОРНИХ РОБІТ

Вимоги щодо оформлення робіт

Звіт до кожної лабораторної роботи оформлюється на окремих аркушах паперу А4 та здається викладачу у встановлені терміни.

Звіт повинен містити наступні елементи :

1). титульну сторінку (рис. 3.1).

2).короткі теоретичні відомості відповідно до теми лабораторної роботи;

3). умови задач;

4). описи алгоритмів розв'язання задач у вигляді блок-схем (для 1 та 2 лабораторної роботи І-ого семестру) та описи створених функцій для решти лабораторних робіт.

5). тексти програм на відповідній мові програмування (Сі або Сі++);

6). список використаної літератури.

До лабораторної роботи додається дискета зі створеними програмами.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Житомирський інженерно-технологічний інститут

кафедра
програмного
забезпечення
обчислювальної техніки

група АК-ХХ

*Лабораторна робота №*Х

Виконав Іваненко І.І.

Перевірив Петренко П.П.

м. Житомир
20ХХ рік

Рис. 3.1. Приклад оформлення титульної сторінки звіту

Нижче наведений приблизний перелік лабораторних робіт, згрупованих за темами.

І семестр (мова програмування Сі)

Лабораторна робота №1
"Прості типи даних. Базові конструкції мови С"

Мета роботи: навчитися складати алгоритми розв'язку задач у вигляді блок-схем, ознайомитися з простими типами даних та базовими конструкціями мови Сі, оволодіти практичними навичками складання, введення, редагування і виконання найпростіших діалогових програм.

Завдання: скласти алгоритми у вигляді блок-схем для задач, які наведено нижче (обрати по три задачі з кожного десятка за правилом, встановленим викладачем) та написати програми для їх розв'язання мовою програмування Сі. Виконати звіт до лабораторної роботи, який вміщує створені блок-схеми та програми.

1. Написати програму, яка буде обчислювати середнє арифметичне та середнє геометричне трьох чисел, що вводяться з клавіатури.
2. Напишіть програму, що визначає належність числа *p*, яке вводиться з клавіатури, до діапазону між *min* та *max*. Значення трьох чисел вводяться користувачем з клавіатури.
3. Напишіть програму для обчислення $min\{a,b,c\}$.
4. Напишіть програму для обчислення $max\{a,b,c\}$.
5. Обчисліть висоту трикутника, якщо відомі його площа та різниця між основою та висотою.
6. Дано три сторони трикутника *a*,*b*,*c*. Визначити його площу та перевірити, чи є він прямокутним.
7. Скласти програму, яка визначає, чи можна побудувати трикутник за заданими довжинами сторін *a*,*b*,*c*; якщо так, визначити, яким він є - гострокутним, прямокутним, різностороннім, рівнобедреним, рівностороннім.
8. З *n* чисел, що вводяться з клавіатури, подайте до друку окремо парні та непарні.
9. Напишіть програму, що знаходить корені звичайного квадратного рівняння за теоремою Вієта.

10. Напишіть програму повного дослідження сукупності коренів біквадратного рівняння. (Якщо коренів не існує, повинно бути виведене відповідне повідомлення, інакше - два або чотири корені.)
11. Знайти найближче ціле до дійсного числа, яке вводиться користувачем з клавіатури.
12. Одержати роздруківку усіх парних чисел від 1 до 1000.
13. Одержати роздруківку усіх непарних чисел від 1 до 1000.
14. Перевірте, чи є введене число з клавіатури простим числом (просте число ділиться тільки на себе і на одиницю).
15. Знайти в першій тисячі натуральних чисел тільки ті числа, що є простими. Вивести їх на екран по одному в кожному рядку.
16. Обрахуйте факторіал числа, що вводиться з клавіатури, коректно передбачивши введення від'ємних чисел.
17. Напишіть програму, що знаходить суму чисел, які передують першому від'ємному числу у введеній послідовності.
18. Користувач вводить числа, закінчуючи введення нулем. Вивести на екран найменше та найбільше число з набору.
19. Користувач вводить числа, закінчуючи введення нулем. Визначити найменше серед додатних та найбільше серед від'ємних.
20. Користувач вводить числа з клавіатури, закінчуючи введення нулем. Визначити наявність у даному наборі від'ємних та додатних чисел. Вивести окремо кількість додатних та від’ємних чисел.
21. Користувач вводить будь-які дійсні числа з клавіатури, закінчуючи введення числом 100. Вивести на екран інформацію про підрахунок у даному наборі як цілих чисел, так і з десятковою комою.
22. Знайти в першій тисячі натуральних чисел тільки ті числа, що без залишку діляться на число, введене користувачем з клавіатури. Вивести їх на екран по три числа в одному рядку.
23. Знайдіть найбільший спільний дільник одночасно не рівних нулю цілих чисел a та b, таких що $a \geq b \geq 0$ (використайте алгоритм Євкліда).
24. Напишіть програму, що друкує у напрямку спадання усі дільники введеного числа.
25. Напишіть програму, що друкує у напрямку зростання усі дільники введеного числа.

26. Знайдіть найменше спільне кратне одночасно не рівних нулю цілих чисел *a* та *b*, таких що $a \geq b \geq 0$.
27. Одержати роздруківку усіх чисел, що закінчуються на цифру 5, з проміжку від 1 до 1000.
28. Одержати роздруківку усіх чисел, що закінчуються на цифру 2, з проміжку від 1 до *N*.
29. Одержати роздруківку усіх чисел, що закінчуються на цифру 3, з проміжку від 1 до *N*.
30. Відшукайте мінімальне та максимальне з десяти чисел, що вводяться з клавіатури (в задачі використайте мінімальну кількість простих змінних).
31. Напишіть програму, що підраховує пробіли, символи табуляції та нового рядка у вхідній послідовності символів, що вводяться з клавіатури.
32. Напишіть програму, що видаляє символ, який визначається користувачем, із вхідного потоку символів, що вводяться. Визначений символ для видалення вводиться з клавіатури на початку роботи програми.
33. Напишіть програму, що перетворює літери, які вводяться з клавіатури, із заголовних у прописні.
34. Написати програму, що підраховує кількість символів пунктуації у рядку символів, що вводиться з клавіатури.
35. Користувач вводить два числа. Визначити, чи ділиться одне число на друге без залишку. Якщо ні, запропонувати найближче ціле, що задовольняє цій умові.
36. Написати програму, що виводить на екран рядкову константу та число, що складається з цифр вашого дня народження, задане у вигляді десяткової, восьмирічної та шістнадцятирічної константи.
37. Перевірте, чи існує чотиризначне натуральне число, куб суми цифр якого дорівнює йому самому.
38. Напишіть програму, що проводить обмін між значеннями двох змінних, не використовуючи при цьому третьої змінної. Запропонуйте декілька варіантів розв'язання такої задачі.
39. Напишіть програму, що підраховує кількість цифр в рядку символів, що вводиться з клавіатури та закінчується точкою.

Лабораторна робота №2
"Цикли та розгалуження. Функції та їх застосування".

Мета роботи: навчитися складати програми циклічних обчислювальних процесів, програми з використанням розгалуження та функцій користувача.

Завдання: Вивести на екран у вигляді таблиці значення функції *F* на інтервалі від $X_{поч}$ до $X_{кін}$ з кроком *H*. Значення *a*, *b*, *c*, $X_{поч}$, $X_{кін}$, *H* – дійсні числа, вводяться з клавіатури. Задачу необхідно розв'язати двома способами: з використанням функцій користувача, та без них. При застосуванні функцій використання глобальних змінних забороняється.

Варіант 1

$$F=\begin{cases} ax^2+b & якщо \quad x<0 \quad і \quad b\neq 0 \\ \dfrac{x-a}{x-c} & якщо \quad x>0 \quad і \quad b=0 \\ \dfrac{x}{c} & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац ЧИ Вц) І (Ац ЧИ Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І і ЧИ — порозрядні.

Варіант 2

$$F=\begin{cases} \dfrac{1}{ax}-b & якщо \quad x+5<0 \quad і \quad c=0 \\ \dfrac{x-a}{x} & якщо \quad x+5>0 \quad і \quad c\neq 0 \\ \dfrac{10x}{c-4} & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац І Вц) ЧИ (Вц І Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І і ЧИ — порозрядні.

Варіант 3

$$F=\begin{cases} ax^2+bx+c & якщо \quad a<0 \quad і \quad c\neq 0 \\ \dfrac{-a}{x-c} & якщо \quad a>0 \quad і \quad c=0 \\ a(x+c) & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз Ац І (Вц ЧИ Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І і ЧИ — порозрядні.

Варіант 4

$$F=\begin{cases} -ax-c & якщо \quad c<0 \quad і \quad x\neq 0 \\ \dfrac{x-a}{-c} & якщо \quad c>0 \quad і \quad x=0 \\ \dfrac{bx}{c-a} & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац ЧИ Вц ЧИ Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операція ЧИ — порозрядна.

Варіант 5

$$F=\begin{cases} a-\dfrac{x}{10+b} & якщо \quad x<0 \quad і \quad b\neq 0 \\ \dfrac{x-a}{x-c} & якщо \quad x>0 \quad і \quad b=0 \\ 3x+\dfrac{2}{c} & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац ЧИ Вц) І Сц не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І і ЧИ — порозрядні.

Варіант 6

$$F=\begin{cases} ax^2+b^2x & якщо \quad c<0 \quad і \quad b\neq 0 \\ \dfrac{x-a}{x+c} & якщо \quad c>0 \quad і \quad b=0 \\ \dfrac{x}{c} & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац І Вц) ЧИ (Ац І Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І і ЧИ — порозрядні.

Варіант 7

$$F=\begin{cases} -ax^2-b & якщо \quad x<5 \quad і \quad c\neq 0 \\ \dfrac{x-a}{x} & якщо \quad x>5 \quad і \quad c=0 \\ \dfrac{-x}{c} & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац ЧИ Вц) МОД2 (Ац ЧИ Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І, ЧИ і МОД2 (додавання за модулем 2) — порозрядні.

Варіант 8

$$F=\begin{cases} -ax^2 & якщо \quad c<0 \quad і \quad a\neq 0 \\ \dfrac{a-x}{cx} & якщо \quad c>0 \quad і \quad a=0 \\ \dfrac{x}{c} & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац МОД2 Вц) І НЕ(Ац ЧИ Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І, ЧИ і МОД2 (додавання за модулем 2) — порозрядні.

Варіант 9

$$F=\begin{cases} ax^2+b^2x & якщо \quad a<0 \quad і \quad x\neq 0 \\ x-\dfrac{a}{x-c} & якщо \quad a>0 \quad і \quad x=0 \\ 1+\dfrac{x}{c} & в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз НЕ(Ац ЧИ Вц) І (Вц ЧИ Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції НЕ, І і ЧИ — порозрядні.

Варіант 10

$$F = \begin{cases} ax^2 - bx + c \quad якщо \quad x < 3 \quad і \quad b \neq 0 \\ \dfrac{x-a}{x-c} \quad якщо \quad x > 3 \quad і \quad b = 0 \\ \dfrac{x}{c} \quad в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз НЕ(Ац ЧИ Вц) І (Ац МОД2 Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції НЕ, І, ЧИ і МОД2 (додавання за модулем 2) — порозрядні.

Варіант 11

$$F = \begin{cases} ax^2 + \dfrac{b}{c} \quad якщо \quad x < 1 \quad і \quad c \neq 0 \\ \dfrac{x-a}{(x-c)^2} \quad якщо \quad x > 15 \quad і \quad c = 0 \\ \dfrac{x^2}{c^2} \quad в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац І Вц) МОД2 Сц не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І і МОД2 (додавання за модулем 2) — порозрядні.

Варіант 12

$$F = \begin{cases} ax^3 + b^2 + c \quad якщо \quad x < 0.6 \quad і \quad b + c \neq 0 \\ \dfrac{x-a}{x-c} \quad якщо \quad x > 0.6 \quad і \quad b + c = 0 \\ \dfrac{x}{c} + \dfrac{x}{a} \quad в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац ЧИ Вц) І Сц не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І і ЧИ — порозрядні.

Варіант 13

$$F = \begin{cases} ax^2 + b \quad якщо \quad x - 1 < 0 \quad і \quad b - x \neq 0 \\ \dfrac{x - a}{x} \quad якщо \quad x - 1 > 0 \quad і \quad b + x = 0 \\ \dfrac{x}{c} \quad в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац ЧИ Вц) МОД2 (Вц І Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції І, ЧИ і МОД2 (додавання за модулем 2) — порозрядні.

Варіант 14

$$F = \begin{cases} -ax^3 - b \quad якщо \quad x + c < 0 \quad і \quad a \neq 0 \\ \dfrac{x - a}{x - c} \quad якщо \quad x + c > 0 \quad і \quad a = 0 \\ \dfrac{x}{c} + \dfrac{c}{x} \quad в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз (Ац МОД2 Вц) ЧИ (Ац МОД2 Сц) не дорівнює нулю, і ціле значення в протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції ЧИ і МОД2 (додавання за модулем 2) — порозрядні.

Варіант 15

$$F = \begin{cases} -ax^2 + b \quad якщо \quad x < 0 \quad і \quad b \neq 0 \\ \dfrac{x}{x - c} \quad якщо \quad x > 0 \quad і \quad b = 0 \\ \dfrac{x}{-c} \quad в \quad інших \quad випадках \end{cases}$$

**Додаткова умова*. Функція *F* повинна приймати дійсне значення, якщо вираз НЕ(Ац ЧИ Вц ЧИ Сц) не дорівнює нулю, і ціле значення в

протилежному випадку. Через Ац, Вц і Сц позначені цілі частини значень *a*, *b*, *c*, операції НЕ і ЧИ — порозрядні.

Лабораторна робота №3 "Одновимірні та багатовимірні масиви, робота з текстовими рядками"

Мета роботи: навчитися складати програми обробки лінійних та двовимірних масивів, реалізовувати найпростіші операції з текстовими рядками.

Завдання: оформити кожен пункт завдання вибраного варіанту у вигляді функції. Всі необхідні дані для функцій передаються їм в якості параметрів. Використання глобальних змінних у функціях не допускається.

Варіант 1

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - суму від'ємних елементів масиву;
   - добуток елементів масиву, що розташовані між максимальним і мінімальним елементами.

   Впорядкувати елементи масиву за зростанням.
2. Дана прямокутна цілочисельна матриця. Визначити:
   - кількість рядків, які не містять жодного нульового елемента;
   - максимальне із чисел, що зустрічається в заданій матриці більше одного разу.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість слів, які мають непарну довжину; виводить на екран частоту входження кожної літери; видаляє текст, що розміщено в круглих дужках.

Варіант 2

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - суму додатних елементів масиву;
   - добуток елементів масиву, що розташовані між максимальним за модулем і мінімальним за модулем елементами.

Впорядкувати елементи масиву за спаданням.

2. Дана прямокутна цілочисельна матриця. Визначити кількість стовпців, які не містять жодного нульового елемента.

   Характеристикою рядка цілочисельної матриці назвемо суму її додатних парних елементів. Переставляючи рядки заданої матриці, розташувати їх у відповідності із зростанням характеристик.

3. З клавіатури вводиться текстовий рядок. Скласти програму, яка перевіряє, чи співпадає кількість відкритих і закритих дужок у введеному рядку (перевірити для круглих та квадратних дужок); виводить на екран найдовше слово; видаляє всі слова, що складаються тільки з латинських літер.

Варіант 3

1. В одновимірному масиві, що складається з N цілих елементів, обчислити:
   - добуток елементів масиву з парними номерами;
   - суму елементів масиву, які розташовані між першим і останнім нульовими елементами.

   Впорядкувати масив таким чином, щоб спочатку розташовувались всі додатні елементи, а потім – всі від'ємні (елементи, рівні 0 вважати додатними).

2. Дана прямокутна цілочисельна матриця. Визначити :
   - кількість стовпців, які містять хоча б один нульовий елемент;
   - номер рядка, в якому знаходиться найдовша серія однакових елементів.

3. З клавіатури вводиться текстовий рядок. Написати програму, яка підраховує кількість різних слів, що входять до заданого тексту; виводить на екран кількість використаних символів; видаляє всі слова, що мають подвоєні літери.

Варіант 4

1. В одномірному масиві, що складається з N дійсних елементів, обчислити:
   - суму елементів масиву з непарними елементами;
   - суму елементів масиву, які розташовані між першим і останнім від'ємними елементами.

Переставити перші M елементів в кінець масиву (M вводиться з клавіатури, M<N).

2. Дана прямокутна цілочисельна матриця. Визначити :
   - добуток елементів в тих рядках, які не містять від'ємних елементів;
   - максимум серед сум елементів діагоналей, паралельних головній діагоналі матриці.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість слів у тексті; виводить на екран слово, що містить найбільшу кількість голосних літер; видаляє з тексту всі непотрібні пробіли.

Варіант 5

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - максимальний елемент масиву;
   - суму елементів масиву, що розташовані до останнього додатного елемента.

   Видалити з масиву всі елементи, модуль яких знаходиться в інтервалі [a,b]. Елементи, які звільняться в кінці масиву заповнити нулями.
2. Дана прямокутна цілочисельна матриця. Визначити :
   - суму елементів в тих стовпцях, які не містять від'ємних елементів;
   - мінімум серед сум модулів елементів діагоналей, паралельних побічній діагоналі матриці.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість розділових знаків у тексті; виводить всі слова, що мають парну кількість літер; міняє місцями першу і останню літери кожного слова.

Варіант 6

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:

- мінімальний елемент масиву;
- суму елементів масиву, що розташовані між першим і останнім додатними елементами.

Перетворити масив таким чином, щоб спочатку розташовувались всі елементи, рівні нулю, а потім – решта.

2. Дана прямокутна цілочисельна матриця. Визначити :
   - суму елементів в тих стовпцях, які містять хоча б один від'ємний елемент;
   - номера рядків і стовпців всіх сідлових точок матриці. Матриця А має сідловий елемент, якщо Аij – мінімальний елемент в і-ому рядку і максимальний в j-му стовпці.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість великих літер у тексті; виводить на екран слова, що мають найменшу кількість літер; видаляє всі слова, що починаються з малої літери.

Варіант 7

1. В одновимірному масиві, що складається з N цілих елементів, обчислити:
   - номер максимального елемента масиву;
   - добуток елементів масиву, що розташовані між першим і другим нульовими елементами.

   Перетворити масив таким чином, щоб в його першій половині розташовувались елементи, що стоять в непарних позиціях, а в другій половині – елементи, що стоять в парних позиціях.
2. Для заданої матриці розміру NxN знайти таке *k*, що *k*-ий рядок матриці співпадає з *k*-м стовпцем. Знайти суму елементів в тих рядках, які містять хоча б один від'ємний елемент.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість чисел у тексті (не цифр, а саме чисел); виводить на екран всі слова, що складаються тільки з латинських літер; видаляє кожне друге слово.

Варіант 8

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - номер мінімального елемента масиву;
   - суму елементів масиву, що розташовані між першим і другим від'ємними елементами.

   Перетворити масив таким чином, щоб спочатку розташовувались всі елементи, модуль яких не перевищує 10, а потім – решта.
2. Характеристикою стовпця цілочисельної матриці назвемо суму модулів його від'ємних непарних елементів. Переставляючи стовпці заданої матриці, розташувати їх у відповідності із ростом характеристик.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість цифр у тексті; виводить на екран слова, що починаються з приголосних літер; знищує всі слова, які починаються і закінчуються за одну й ту ж літеру.

Варіант 9

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - максимальний за модулем елемент масиву;
   - суму елементів масиву, що розташовані між першим і другим додатними елементами.

   Перетворити масив таким чином, щоб всі елементи, рівні нулю та одиниці, розташовувались після всіх інших.
2. Коефіцієнти системи лінійних рівнянь задані у вигляді прямокутної матриці. За допомогою допустимих перетворень звести матрицю до трикутного вигляду. Знайти кількість рядків, середнє арифметичне елементів яких менше заданої величини.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість слів у тексті, які закінчуються на голосну літеру; виводить на екран всі слова, довжина яких менша п'яти символів; видаляє всі слова, які містять хоча б одну латинську літеру.

Варіант 10

1. В одновимірному масиві, що складається з N цілих елементів, обчислити:
   - мінімальний за модулем елемент масиву;
   - суму модулів елементів масиву, розташованих після першого елемента, рівного нулю.

   Перетворити масив таким чином, щоб в першій його половині розташовувались елементи, що стоять на парних позиціях, а в другій половині – елементи, що стоять в непарних позиціях.
2. Здійснити циклічний зсув елементів прямокутної матриці на n елементів вправо або вниз (в залежності від введеного режиму).
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість слів у тексті, які починаються з голосної літери; виводить на екран всі слова, які містять непарну кількість приголосних літер; видаляє всі числа з тексту.

Варіант 11

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - мінімальний за модулем елемент масиву;
   - суму модулів елементів, які розташовані після першого від'ємного елемента.

   Стиснути масив, видаливши з нього всі елементи, величина яких знаходиться на інтервалі [a,b]. Місце, які звільниться в кінці масиву заповнити нулями.
2. Дана цілочисельна прямокутна матриця. Визначити номер першого з стовпців, які містять хоча б один нульовий елемент. Характеристикою рядка цілочисельної матриці назвемо суму її від'ємних парних елементів. Переставляючи рядки заданої матриці, розташувати їх у відповідності зі спаданням характеристик.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка замінює всі великі літери, що входять до тексту на відповідні малі; виводить на екран найдовше слово; видаляє всі слова, що містять непарну кількість приголосних літер.

Варіант 12

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - суму індексів додатних елементів;
   - суму модулів елементів, які розташовані після першого додатного елемента.

   Перетворити масив таким чином, щоб спочатку розташовувались всі елементи, ціла частина яких лежить в інтервалі [a,b], а потім – решта.
2. Впорядкувати рядки цілочисельної прямокутної матриці за зростанням кількості однакових елементів в кожному рядку. Знайти номер першого із стовпців, який не містить жодного від'ємного елемента.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка підраховує кількість слів, які містять однакову кількість голосних і приголосних літер; виводить на екран найдовше слово; видаляє з тексту всі слова-паліндроми.

Варіант 13

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - кількість елементів масиву, рівних нулю;
   - суму елементів масиву, які лежать в діапазоні від A до B.

   Впорядкувати елементи масиву за спаданням модулів елементів.
2. Дана цілочисельна прямокутна матриця. Визначити:
   - кількість рядків, які містять хоча б один нульовий елемент;
   - номер стовпця, в якому знаходиться найдовша серія однакових елементів.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка виводить на екран всі символи, які розташовані після першого символу „:"; підраховує кількість речень, що містять непарну кількість слів; видаляє з тексту всі слова, які розташовані після ком.

Варіант 14

1. В одновимірному масиві, що складається з N дійсних елементів, обчислити:
   - кількість елементів масиву, більших С;
   - добуток елементів масиву, що розташовані після мінімального елемента .

   Впорядкувати елементи масиву за зростанням модулів елементів.
2. Дана цілочисельна прямокутна матриця. Визначити:
   - кількість від'ємних елементів в тих рядках, які містять хоча б один нульовий елемент;
   - суму модулів елементів, які розташовані після першого додатного елемента
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка рахує кількість слів у кожному реченні; виводить на екран найдовше речення; видаляє всі слова, передостання літера яких голосна.

Варіант 15

1. В одновимірному масиві, що складається з N цілих елементів, обчислити:
   - номер елемента масиву, найближчого до середнього арифметичного його значень;
   - сума елементів масиву, що розташовані між першим від'ємним та другим додатним елементами.

   Перетворити масив таким чином, щоб в його першій половині розташовувались елементи, що стоять в парних позиціях, а в другій половині – елементи, що стоять в непарних позиціях.
2. Для заданої матриці розміру NxN знайти такі *k* та *n*, що сума елементів *k*-стовпця матриці співпадає з сумою елементів *n*-го рядка. Знайти суму елементів в тих рядках, які містять хоча б два ненульових елементи.
3. З клавіатури вводиться текстовий рядок. Скласти програму, яка інвертує рядок, подаючи його у зворотному вигляді; підраховує кількість чисел у тексті; видаляє всі слова, що починаються з голосних літер.

Лабораторна робота №4 "Структури та їх використання. Масиви структур. Використання динамічної пам'яті"

Мета роботи : оволодіти практичними навичками використання структур та масивів структур, навчитися складати програми для виконання операцій з полями структур, навчитися використовувати динамічне виділення пам'яті.

Завдання :

Варіант 1

1. Описати структуру з ім'ям STUDENT, яка містить наступні поля:
   - NAME – прізвище та ініціали;
   - GROUP – номер групи;
   - SES – оцінки з п'яти предметів (масив з п'яти елементів).

Написати програму, що реалізовує наступні дії окремими функціями:

- введення з клавіатури даних в масив STUD, що складається з N змінних типу STUDENT;
- впорядкування записів за зростанням значень поля GROUP;
- виведення на екран прізвищ і номерів груп для всіх студентів, середній бал яких більший за 4.0; якщо таких студентів немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 5).

Варіант 2

1. Описати структуру з ім'ям ABITURIENT, яка містить наступні поля:
   - NAME – прізвище, ініціали;
   - GENDER - стать;
   - SPEC – назва спеціальності;
   - EXAM – результати вступних іспити з трьох предметів (масив з трьох елементів).

Написати програму, що окремими функціями реалізовує наступні дії:

- введення з клавіатури даних в масив ABITUR, що складається з N змінних типу ABITURIENT;
- впорядкування записів за зростанням середнього бала;

- виведення на екран прізвищ та назв спеціальностей для всіх абітурієнтів, що мають бал нижче, ніж прохідний, який визначається користувачем програми; якщо таких студентів немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 7).

Варіант 3

1. Описати структуру з ім’ям SCHOOL, яка містить наступні поля:

- NAME – прізвище та ім'я учня;
- GROUP – номер групи;
- SUBJECT – успішність з п'яти предметів (масив з п’яти елементів).

Написати програму, що окремими функціями виконує наступні дії:

- введення з клавіатури даних в масив LEARNER, що складається з N змінних типу SCHOOL;
- впорядкування записів за алфавітом;
- виведення на екран прізвищ і номерів груп для всіх студентів, що мають хоча б одну оцінку 2; якщо таких студентів немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 10).

Варіант 4

1. Описати структуру з ім’ям AEROFLOT, яка містить наступні поля:

- CITY – назва населеного пункту призначення;
- NUM – номер рейса;
- TYPE – тип літака.

Написати програму, що окремими функціями реалізовує наступні дії:

- введення з клавіатури даних в масив AIR, що складається з N змінних типу AEROFLOT;
- впорядкування записів за зростанням номеру рейсу;
- виведення на екран номерів рейсів і типів літаків, що вилетіли в пункт призначення, назва якого співпала з назвою, введеною

з клавіатури; якщо таких рейсів немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 14).

Варіант 5

1. Описати структуру з ім'ям SKLAD, яка містить наступні поля:
   - NAME – назва товару;
   - TYPE – одиниця виміру товару;
   - QUANTITY – кількість одиниць товару;
   - COST – ціна одиниці товару;

Написати програму, що окремими функціями виконує наступні дії:

- введення з клавіатури даних в масив SHOP, що складається з N змінних типу SKLAD;
- впорядкування записів за назвами товару;
- виведення на екран інформації про товар, його кількість, ціну одиниці та обчислену загальну суму на складі, назва якого вводиться з клавіатури; якщо такого немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 12).

Варіант 6

1. Описати структуру з ім'ям WORKER, яка містить наступні поля:
   - NAME – прізвище та ініціали працівника;
   - POS – назва посади;
   - YEAR – рік прийняття на роботу;
   - MONTH - місяць прийняття на роботу.

Написати програму, що окремими функціями виконує наступні дії:

- введення з клавіатури даних в масив TABL, що складається з N змінних типу WORKER;
- впорядкування записів в алфавітному порядку;
- виведення на екран прізвищ працівників, стаж роботи яких перевищує значення, введене з клавіатури; якщо таких працівників немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 1).

Варіант 7

1. Описати структуру з ім’ям TRAIN, яка містить наступні поля:
   - NAZV – назва пункту призначення;
   - NUMR – номер потягу;
   - DATE – дата відправлення;
   - TIME – час відправлення.

   Написати програму, що окремими функціями виконує наступні дії:
   - введення з клавіатури даних в масив RASP, що складається з N змінних типу TRAIN;
   - впорядкування записів за алфавітом за назвами пунктів призначення;
   - виведення на екран інформації про поїзди, що відправляються після введеного з клавіатури дня та часу; якщо таких поїздів немає, то вивести відповідне повідомлення.
2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 2).

Варіант 8

1. Описати структуру з ім’ям TIMETABLE, яка містить наступні поля:
   - NAZV – назва пункту призначення;
   - NUMR – номер поїзда;
   - DATE – дата відправлення;
   - TIME – час відправлення.

   Написати програму, що окремими функціями виконує наступні дії:
   - введення з клавіатури даних в масив TRAIN, що складається з N змінних типу TIMETABLE;
   - впорядкування записів за датою та часом відправлення поїзда;
   - виведення на екран інформації про поїзди, що направляються в пункт призначення, назва якого введена з клавіатури; якщо таких поїздів немає, то вивести відповідне повідомлення.
2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 13).

Варіант 9

1. Описати структуру з ім'ям TIMETABLE, яка містить наступні поля:
   - NAZV – назва пункту призначення;
   - NUMR – номер поїзда;
   - DATE – дата відправлення;
   - TIME – час відправлення.

   Написати програму, що окремими функціями виконує наступні дії:
   - введення з клавіатури даних в масив TRAIN, що складається з N структур типу TIMETABLE;
   - впорядкування записів за номерами поїздів;
   - виведення на екран інформацію про поїзди, дата відправлення яких введена з клавіатури; якщо таких поїздів немає, то вивести відповідне повідомлення.
2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 3).

Варіант 10

1. Описати структуру з ім'ям ITINERARY, яка містить наступні поля:
   - FIRST – назва початкового пункту маршруту;
   - FINAL – назва кінцевого пункту маршруту;
   - NUM – номер маршруту.
   - DISTANCE – відстань у кілометрах.

   Написати програму, що окремими функціями виконує наступні дії:
   - введення з клавіатури даних в масив ROUT, що складається з N змінних типу ITINERARY;
   - впорядкування записів за спаданням відстані у кілометрах;
   - виведення на екран інформації про маршрут, номер якого введений з клавіатури; якщо таких маршрутів немає, то вивести відповідне повідомлення.
2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 8).

Варіант 11

1. Описати структуру з ім'ям ITINERARY, яка містить наступні поля:

- BEG – назва початкового пункту маршруту;
- END – назва кінцевого пункту маршруту;
- NUM – номер маршруту;
- DISTANCE – відстань у кілометрах.

Написати програму, що окремими функціями виконує наступні дії:

- введення з клавіатури даних в масив ROUT, що складається з N змінних типу ITINERARY;
- впорядкування записів за номерами маршрутів;
- виведення на екран інформацію про маршрути, які починаються або закінчуються в пункті, назва якого введена з клавіатури; якщо таких маршрутів немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 4).

Варіант 12

1. Описати структуру з ім’ям NOTE, яка містить наступні поля:

- NAME – прізвище, ім’я;
- TEL – номер телефону;
- BDAY – день народження (масив із трьох чисел).

Написати програму, що окремими функціями виконує наступні дії:

- введення з клавіатури даних в масив BLOCKNOTE, що складається з N змінних типу NOTE;
- впорядкування записів за зростанням дат днів народження;
- виведення на екран інформації про людей, чиї дні народження припадають на місяць, значення якого введено з клавіатури; якщо таких людей немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 6).

Варіант 13

1. Описати структуру з ім’ям ZNAK, яка містить наступні поля:

- NAME – прізвище, ім’я;
- ZODIAC – знак Зодіаку;
- BDAY – день народження (масив із трьох чисел).

Написати програму, що окремими функціями виконує наступні дії:

- введення з клавіатури даних в масив BOOK, що складається з N змінних типу ZNAK;
- впорядкування записів за спаданням дат народження;
- виведення на екран інформації про людину, чиє прізвище введене з клавіатури; якщо таких людей немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 11).

Варіант 14

1. Описати структуру з ім'ям ABITURIENT, яка містить наступні поля:

- NAME – прізвище, ініціали;
- GENDER - стать;
- SPEC – назва спеціальності;
- EXAM – результати вступних іспитів з трьох предметів (масив з трьох елементів).

Написати програму, що окремими функціями виконує наступні дії:

- введення з клавіатури даних в масив ABIT, що складається з N змінних типу ABITURIENT;
- впорядкування записів за алфавітом;
- виведення на екран прізвищ та назв спеціальностей для всіх абітурієнтів, що набрали прохідний бал, який визначається користувачем програми; якщо таких студентів немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 15).

Варіант 15

1. Описати структуру з ім'ям TOVAR, яка містить наступні поля:

- NAME – назва товару;
- TYPE – одиниця виміру товару;
- SORT – сорт товару;
- QUANTITY – кількість одиниць товару;
- COST – ціна одиниці товару;

Написати програму, що окремими функціями виконує наступні дії:

- введення з клавіатури даних в масив SHOP, що складається з N змінних типу TOVAR;
- впорядкування записів за спаданням кількості одиниць товару;
- виведення на екран інформації про товар, його кількість, ціну одиниці та обчислену загальну суму на складі; назва товару вводиться з клавіатури, якщо його немає, то вивести відповідне повідомлення.

2. Виконати завдання №2 з попередньої лабораторної роботи №3 використовуючи динамічне виділення пам'яті (варіант 9).

Лабораторна робота №5
"Робота з файлами. Обробка текстової інформації."

Мета : навчитися складати програми для роботи з файлами заданої структури та виконання обробки текстової інформації.

Завдання:

В *завданні №1* необхідно написати програму, яка виконує вказані операції (кожну операцію оформити окремою функцією) з інформацією, що знаходиться у текстовому файлі *input.txt* і записує всі результати роботи програми у файл *output.txt*. Скласти блок-схему для алгоритму розв'язку задачі.

Вхідний файл : input.txt

Вихідний файл : output.txt

В *завданні №2* необхідно організувати файл даних з вказаною нижче структурою та передбачити функції, які дозволяють :

- коригування обраного запису файлу;
- пошук інформації за різними полями;
- додавання записів у кінець бази даних;
- вилучення інформації з бази даних.

Варіант 1

1. Написати програму, яка копіює вміст вхідного файла у вихідний файл; підраховує кількість чисел у тексті (не цифр, а саме чисел); виділяє всі слова, що складаються тільки з латинських літер; видаляє кожне друге слово.

2. Структура з ім'ям SKLAD, яка містить наступні поля:

- NAME – назва товару;
- TYPE – одиниця виміру товару;
- QUANTITY – кількість одиниць товару;
- COST – ціна одиниці товару;

Варіант 2

1. Написати програму, яка копіює вміст вхідного файла у вихідний файл; підраховує кількість слів у тексті, які починаються з голосної літери; знаходить всі слова, які містять непарну кількість приголосних літер; видаляє всі числа з тексту.

2. Структура з ім'ям ABITURIENT, яка містить наступні поля:

- NAME – прізвище, ініціали;
- GENDER – стать;
- SPEC – назва спеціальності;
- EXAM – результати вступних іспитів з трьох предметів (масив з трьох елементів).

Варіант 3

1. Написати програму, яка копіює вміст вхідного файла у вихідний; інвертує рядок, подаючи його у зворотному вигляді; підраховує кількість чисел у тексті; видаляє всі слова, що починаються з голосних літер.

2. Структура з ім'ям TOVAR, яка містить наступні поля:

- NAME – назва товару;
- TYPE – одиниця виміру товару;
- SORT – сорт товару;
- QUANTITY – кількість одиниць товару;
- COST – ціна одиниці товару;

Варіант 4

1. Написати програму, яка копіює вміст вхідного файла у вихідний; замінює всі великі літери, що входять до тексту на відповідні малі; визначає найдовше слово; видаляє всі слова, що містять непарну кількість приголосних літер.

2. Структура з ім'ям ITINERARY, яка містить наступні поля:

- FIRST – назва початкового пункту маршруту;
- FINAL – назва кінцевого пункту маршруту;
- NUM – номер маршруту.
- DISTANCE – відстань у кілометрах.

Варіант 5

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів, які мають непарну довжину; виводить на екран частоту входження кожної літери у тексті; перевіряє правильність розташування круглих дужок у тексті; видаляє всі парні числа з тексту.

2. Структура з ім'ям ABONENT, яка містить наступні поля:

- NAME – прізвище абонента;
- INIT – ініціали абонента;
- NOMER – номер телефону;
- ADRESS – домашня адреса.

Варіант 6

1. Написати програму, яка копіює вміст вхідного файла у вихідний; перевіряє, чи співпадає кількість відкритих і закритих дужок у введеному рядку (перевірити для круглих та квадратних дужок); знаходить найдовше слово; видаляє всі слова, що складаються тільки з латинських літер.

2. Структура з ім'ям AEROFLOT, яка містить наступні поля:

- NAZV – назва пункту призначення;
- NUMR – номер літака;
- TYPE – тип літака;
- TIME – час відправлення.

Варіант 7

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів непарної довжини; замінює всі слова, записані кирилицею на аналогічні, записані латиницею (обробка – obrobka); видаляє всі слова, які починаються і закінчуються на голосну літеру.

2. Структура з ім'ям ABONENT, яка містить наступні поля:

- NAME – прізвище та ініціали користувача мережі;
- LOGIN – обліковий запис;
- PASSWORD – пароль;
- TYPE – тип облікового запису;
- DATE – рік та місяць прийняття на роботу;

Варіант 8

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість різних слів, що входять до заданого тексту; визначає кількість використаних символів; видаляє всі слова, що мають подвоєні літери.

2. Структура з ім'ям STUDENT, яка містить наступні поля:

- NAME – прізвище та ініціали;
- DATABIRTH – дата народження;
- GROUP – номер групи;
- SES – успішність з п'яти предметів (масив з п'яти елементів).

Варіант 9

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів у тексті; виділяє слово, що містить найбільшу кількість голосних літер; видаляє з тексту всі непотрібні пробіли.

2. Структура з ім'ям NOTE, яка містить наступні поля:

- NAME – прізвище, ім'я;
- TEL – шестизначний номер телефону;
- BDAY – день народження (масив із трьох чисел).

Варіант 10

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість розділових знаків у тексті; виводить всі слова, що мають парну кількість літер; міняє місцями першу і останню літери кожного слова.

2. Структура з ім'ям DETAL, яка містить наступні поля:

- NAME – назва деталі;
- SORT – сорт виробу;
- DATE – дата виготовлення (масив із трьох чисел).

- QUANT – кількість;
- COST – ціна одиниці.

Варіант 11

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість цифр у тексті; визначає слова, що починаються з приголосних літер; знищує всі слова, які починаються і закінчуються за одну й ту ж літеру.

2. Структура з ім'ям TOVAR, яка містить наступні поля:

- NAME – назва товару;
- TYPE – одиниця виміру товару;
- SORT – сорт товару;
- QUANTITY – кількість одиниць товару;
- COST – ціна одиниці товару;

Варіант 12

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість великих літер у тексті; визначає слова, що мають найменшу кількість літер; видаляє всі слова, що починаються з малої літери.

2. Структура з ім'ям TRAIN, яка містить наступні поля:

- NAZV – назва пункту призначення;
- NUMR – номер поїзда;
- DATE – дата відправлення;
- TIME – час відправлення.

Варіант 13

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів, які містять однакову кількість голосних і приголосних літер; визначає найдовше слово; видаляє з тексту всі слова-паліндроми..

2. Структура з ім'ям SKLAD, яка містить наступні поля:

- NAME – назва товару;
- TYPE – одиниця виміру товару;

- QUANTITY – кількість одиниць товару;
- COST – ціна одиниці товару;

Варіант 14

1. Написати програму, яка копіює вміст вхідного файла у вихідний; виводить на екран всі символи, які розташовані після першого символу „:”; підраховує кількість речень, що містять непарну кількість слів; видаляє з тексту всі слова, які розташовані після ком.

2. Структура з ім’ям ITINERARY, яка містить наступні поля:

- FIRST – назва початкового пункту маршруту;
- FINAL – назва кінцевого пункту маршруту;
- NUM – номер маршруту.
- DISTANCE - відстань у кілометрах.

Варіант 15

1. Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів у тексті, які закінчуються на голосну літеру; знаходить всі слова, довжина яких менша п’яти символів; видаляє всі слова, які містять хоча б одну латинську літеру.

2. Структура з ім’ям TRAIN, яка містить наступні поля:

- NAZV – назва пункту призначення;
- NUMR – номер поїзда;
- DATE – дата відправлення;
- TIME – час відправлення.

Задачі на складання ефективних алгоритмів

Краще підготовлені студенти можуть обирати (за узгодженням з викладачем) задачі з даного розділу замість тих, що пропонуються для основної групи студентів. Завдання згруповані за відповідними номерами лабораторних робіт першого семестру - №1, №3 та №5, та відповідають їх тематиці. Деякі з цих задач, або їх елементи пропонувалися на різних олімпіадах з програмування.

Л.1.1. На площині задано координати двох протилежних вершин квадрата. Необхідно знайти координати двох інших його вершин, якщо сторони квадрата паралельні осям координат.

Л.1.2. Скласти програму скорочення дробу виду $\frac{m}{n}$, де *m*, *n*, - натуральні числа.

Л.1.3. Задано два цілих числа *a* і *b*, причому *a* не дорівнює *b*. Знайти серед них менше, не використовуючи операторів вибору, умовних операторів, циклів. Дозволяється використовувати тільки арифметичні операції.

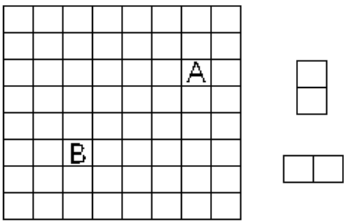
Л.1.4. На інтервалі [1000;9999] знайти всі прості числа для кожного з яких сума першої та другої цифри рівна сумі третьої та четвертої цифри.

Л.1.5. Людина піднімається по сходах, ступаючи на наступну сходинку, або перестрибуючи через одну чи дві сходинки. Знайти, скількома способами вона зможе піднятися на *N*-у сходинку. Масивів не використовувати.

Л.1.6. Переставити цифри числа *N* так, щоб одержати найбільш можливе число, вивести його на екран. Реалізувати алгоритм без використання масивів.

Л.1.7. Знайти всі трьохзначні числа, рівні сумі факторіалів своїх цифр.

Л.1.8. Нехай $A_1...A_N$ - послідовність цілих чисел. Позначимо *максимальний* та *мінімальний* елементи цієї послідовності як *max* та *min* відповідно. Обчислимо суму елементів цієї послідовності $S = A_1 + A_2 + \dots A_N$. Замінимо кожний елемент послідовності на різницю *S* та цього елемента: $A_i = S - A_i$. Таку дію повторимо *K* разів. Написати програму, яка за послідовністю A_I (вводиться з клавіатури), отриманою в результаті *K*-кратного повторення цієї операції обчислює різницю *max* - *min*. Масивів не використовувати.



Л.1.9. В квадратній дошці розміром NxN клітинок вирізано дві клітинки з координатами $A(x_1,y_1)$, $B(x_2,y_2)$. Визначити чи можна повністю покрити дощечками розміром 1x2 всі клітинки дошки. Координати вирізаних клітинок вводяться з клавіатури. Масиви не використовувати

Л.1.10. Надрукувати всі числа від 1 до N^2 у вигляді квадратної таблиці розміром NxN, розташувавши їх за наступною схемою без використання масиву :

1 3 6 10
2 5 9 13
4 8 12 15
7 11 14 16

Л.1.11. Клієнт банка забув чотиризначний шифр свого сейфа, але пам'ятав, що цей шифр - просте число, добуток його цифр рівний *N*. Яка найменша кількість спроб гарантує йому відкриття сейфу. На екран вивести всі необхідні спроби та їх кількість.

Л.1.12. Обчислити кількість чисел в системі числення з основою *K*, які містять ≤ *N* знаків, таких, що їх запис не містить двох підряд розміщених нулів. Числа *N* і *K* вводяться з клавіатури. Масивів не використовувати.

Л.3.1. У масиві A[1..N] кожний елемент рівний 0,1,2 або 3. Не використовуючи додаткової таблиці, переставити елементи масиву так, щоб спочатку розташовувались усі трійки, потім двійки, одиниці, врешті, всі нулі. Число *N* та всі елементи масиву вводяться з клавіатури.

Л.3.2. Дано рядок (вводиться з клавіатури), що містить шлях до файла або каталогу, записаний за згодами, прийнятими в MS DOS. Перетворити даний рядок таким чином, щоб він містив шлях в форматі ОС Unix.

Л.3.3. Скласти функцію для підрахунку кількості різних чисел у масиві, що містить *N* елементів.

Л.3.4. Паліндромом називається симетричний рядок, який однаково читається як зліва направо, так і справа наліво. Потрібно написати функцію, яка буде визначати, чи є введений рядок паліндромом.

Л.3.5. Користувачу, що зареєструвався на FTP-сервері для отримання доступу до файлів на ньому потрібно набрати в FTP-браузері команду вигляду: ftp://логін:пароль@адреса_сервера. Написати програму, яка з введеного рядка виділяє логін, пароль, адресу FTP-сервера і друкує цю інформацію на екран.

Формат вхідних даних :
ftp://username:parol@ftp.server.ua

Формат вихідних даних :
Адреса сервера: ftp.server.ua
Логін: username
Пароль: parol

Л.3.6. Дано ігрове поле розміром MxN клітинок. У верхній лівій клітинці (клітинка старту) знаходиться фішка. У нижньому правому кутку знаходиться клітинка фінішу. Фішку дозволяється рухати вниз або вправо на будь-яку кількість клітинок. Написати програму, яка за числами *M* і *N* буде обчислювати кількість шляхів, якими можна перемістити фішку з клітинки старту в клітинку фінішу. Для прикладу, зображеного на малюнку (M=2, N=4) відповідь 4.

Л.3.7. За заданими координатами вершин многокутника перевірити, чи є він опуклим. Кількість вершин та координати вводяться з клавіатури. *Примітка :* координати вершин не обов'язково впорядковані за порядком обходу.

Л.3.8. Задано ціле додатне число *N* (N≤1000000000). Записати це число словами у вигляді рядкової величини.

Наприклад : 1024 - тисяча двадцять чотири
2 - два.

Л.3.9. Написати функцію, яка буде обчислювати визначник матриці A[NxN]. Передбачити можливість введення елементів матриці з клавіатури та можливість заповнення матриці випадковими числами.

Л.3.10. Магічним квадратом порядку *N* називається таке розташування цілих чисел у матриці NxN, що суми елементів у кожному рядку, кожному стовпці і двох діагоналях співпадають. Скласти функцію для побудови магічного квадрату порядку *N* (*N* - непарне число), використовуючи числа від 1 до N^2 . Число *N* вводиться з клавіатури.

8	1	6
3	5	7
4	9	2

Л.3.11. Написати програму, яка буде обчислювати всі цифри n! при $n \le 100$.

Л.3.12. Скласти програму, яка буде обчислювати значення $2^{64} - 1$ зі збереженням всіх цифр.

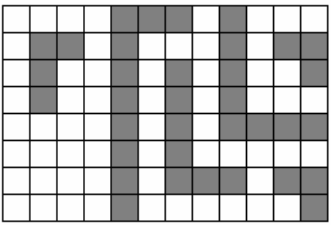
Л.5.1. Дано текстовий файл, в якому містяться різні слова довжиною від одного та більше символів, відокремлені довільним числом пробілів. Побудувати частотний словник слів тексту у вигляді підсумкової таблиці з двох колонок, вказавши, скільки разів зустрічається кожне із слів тексту.

Л.5.2. Дано файл, в якому зустрічаються теги та . Замінити кожне входження "" на "<курсив>", а кожне входження "" на "<кінець курсиву>".

Примітка : в програмі передбачити, що літера "i" може бути як малою, так і великою. Вхідні дані зчитуються з файла LAB5_1.TXT і записуються у файл LAB5_1.OUT.

Л.5.3. У прямокутнику розміром MxN, розбитому на одиничні клітинки міститься "змійка" (неперервна ламана лінія шириною в одну клітинку, яка може згинатися лише на 90 градусів). "Змійка" може утворювати замкнутий або розімкнутий контур. "Змійка" себе не перетинає і ніде не дотикається різними частинками. Інформація про розташування "змійки" задається матрицею A [N;M]. Значення A[i;j]==1, якщо клітинка належить "змійці" і A[i,j]==0, якщо не належить. Значення *M, N* та матриця зчитується з файла LAB5_2.TXT, а результати записуються у файл LAB5_2.OUT. Визначити, чи утворює "змійка" замкнутий контур.

Приклад вхідного файла:
6 16
1 1 1 1 1 1 0 0 0 1 1 1 1 1 0 0
1 0 0 0 0 1 1 1 1 1 0 0 0 1 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1
0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
Приклад вихідного файла:
Змійка утворює замкнутий контур.



Л.5.4. У прямокутнику розміром MxN, розбитому на одиничні клітинки містяться "кутики". "Кутик" - смужка товщиною 1 клітинку, зігнута у довільному місці на 90 градусів. "Кутики" ніде не накладаються і ніде не дотикаються. Інформація про розташування кутиків задається прямокутною матрицею A[N;M]. Значення A[i;j]==1 означає, що клітинка належить деякому кутику, а A[i;j]==0 означає, що клітинка не належить кутику. Вхідні дані *M, N*, матриця *A* зчитуються з файла LAB5_3.TXT. Написати програму, яка визначає кількість кутиків. Результати записати у файл LAB5_3_OUT.

Приклад вхідного файла :

```
8 12
0 0 0 0 1 1 1 0 1 0 0 0
0 1 1 0 1 0 0 0 1 0 1 1
0 1 0 0 1 0 1 0 1 0 0 1
0 1 0 0 1 0 1 0 1 0 0 0
0 0 0 0 1 0 1 0 1 1 1 1
0 0 0 0 1 0 1 0 0 0 0 0
0 0 0 0 1 0 1 1 1 0 1 1
0 0 0 0 1 0 0 0 0 0 0 1
```

Приклад вихідного файла:

```
Всього 6 кутиків.
```

Л.5.5. Організувати файл бази даних на вільну тему (ім'я файла - LAB5_5.DAT). Передбачити можливість редагування інформації в БД. Організувати пошук інформації за кількома полями.

Л.5.6. Дано рядок, що складається із слів, розділених пробілом. Написати програму, що знищує зайві пробіли. Пробіл вважається зайвим, якщо він стоїть на початку рядка, стоїть в кінці рядка, або слідує за пробілом. Рядок зчитується з файла LAB5_6.TXT, а результати записуються у файл LAB5_6.OUT.

Л.5.7. У файлі знаходиться текст програми на мові Сі. Потрібно написати програму, що буде видаляти коментарі в тексті програми. Як відомо, коментар - це послідовність символів, які знаходяться між "/*" і "*/". Коментар може бути багаторядковим, тобто починатися в одному рядку, а закінчуватися в іншому рядку. Вхідні дані необхідно прочитати з файла LAB5_7.TXT, а результати записати у файл LAB5_7.OUT.

Л.5.8. Напишіть програму, що здійснює перенесення занадто довгих рядків. Слова розбивати не можна (слово, яке не можна розмістити, варто перенести цілком на новий рядок). Ширина рядка дорівнює 80. Вхідні дані прочитати з файла LAB5_8.TXT, результати записати у файл LAB5_8.OUT.

Л.5.9. Задано шаблон з круглих дужок і знаків запитання. Потрібно визначити, скількома способам можна замінити знаки запитання круглими дужками так, щоб вийшов правильний вираз із дужок. Перший рядок файла LAB5_9.TXT містить заданий шаблон. У вихідний файл LAB5_9.OUT вивести знайдену кількість способів.

Приклад вхідного файла.

????(?

Приклад вихідного файла.

2

Л.5.10. Шахова асоціація вирішила обладнати всіх своїх співробітників такими телефонними номерами, які набиралися б на кнопочному телефоні ходом шахового коня. Наприклад, ходом шахового коня набирається телефон 340-49-27. При цьому телефонний номер не може починатися ні з цифри 0, ні з цифри 8. Написати програму, яка визначає кількість телефонних номерів довжини N, які набираються ходом шахового коня.

7 8 9
4 5 6
1 2 3
  0

Приклад вхідних даних (LAB5_10.TXT):

2

Приклад вихідних даних (LAB5_10.OUT):

16

Л.5.11. Під час друку великих документів може виникнути потреба друкувати не весь документ, а тільки деякі його сторінки. Серед аргументів програми друку є рядок з послідовністю номерів сторінок. Потрібно надрукувати не окремі сторінки, а діапазони сторінок і, можливо, вказувати початок і кінець діапазонів, а не послідовні числа.

Завдання:

Напишіть програму, яка буде перетворювати списки сторінок у відповідну послідовність номерів сторінок.

Приклад вхідних даних (LAB5_11.TXT):

1,4-5,7-7,10-20

Приклад вихідних даних (LAB5_11.OUT):

1,4,5,7,10,11,12,13,14,15,16,17,18,19,20

Л.5.12. Прямокутник, сторони якого виражені натуральними числами *a* і *b*, розділений на квадрати розміром 1x1. Знайти число квадратів, які перетинає діагональ прямокутника.

Вхідні дані : LAB5_12.TXT
Вихідні дані : LAB5_12.OUT

Л.5.13. У файлі STATE.DAT міститься деяка кількість назв міст (по одній в кожному рядку). Утворіть з даного набору слів замкнений ланцюжок, в якому кожне наступне слово починається з літери, якою закінчувалося попереднє, використавши найбільшу кількість слів. Всі слова у файлі різні і у ланцюжку можна використовувати не більше одного разу. Програма STATE.C повинна на екран та у перший рядок файлу STATE.SOL вивести кількість використаних слів, а далі – всі використані слова у потрібній послідовності (по одному слову в кожному рядку). У випадку, коли ланцюжок утворити неможливо, у файл STATE.SOL необхідно записати лише одне число 0.

Приклад вхідних та вихідних даних.

STATE.DAT	*STATE.SOL*
МОСКВА	5
ВАРШАВА	ПАРИЖ
ПАРИЖ	ЖИТОМИР
ЖИТОМИР	РИМ
МУРМАНСЬК	МУРМАНСЬК
КОНОТОП	КОНОТОП
РИМ	

Л.5.14. Написати програму-архіватор, яка буде перетворювати інформацію, що записана у файлі таким чином, щоб вона займала якомога менший розмір та програму, яка відновлює початковий файл за архівним.

Додаткові задачі, що пропонувалися на Всеукраїнських олімпіадах з програмування у 2001 та 2002 роках (м. Одеса, м. Чернівці)

Задача “Шифр”

Задано символьний рядок S довжини N ($0 \le N \le 100$) та словник, що містить M слів ($0 \le M \le 100$), довжина кожного з яких не перебільшує N. Слова словника і рядок S складаються з літер `a`, `b`, …, `z`.

Завдання

Складіть програму CIPHER, котра визначає найменшу кількість символів, яку треба викреслити із заданого рядка S, щоб результуючий рядок можна було подати як послідовність слів словника. Кількість використань кожного слова не обмежується. Вважається, що пустий рядок можна подати за допомогою слів будь-якого словника.

Рядок у прикладі після викреслювання зайвих букв **f** і **t** набуде вигляду **abachdsya** (було зроблено два викреслювання: **abafchtdsya**), та може бути поданий як послідовних наступних слів: **a, bach, dsy, a**.

Вхідні дані

В першому рядку вхідного файла CIPHER.DAT знаходяться два цілих числа *N* та *M*, відокремлених пропусками. У другому рядку знаходиться символьний рядок *S*. У кожному з наступних *M* рядків знаходиться слово словника.

Приклад вхідного файлу

```
11 5
abafchtdsya
aba
a
bach
dsy
zero
```

Вихідні дані

В єдиному рядку вихідного файлу CIPHER.SOL має знаходитись натуральне число – мінімальна кількість викреслювань, після яких зашифрований рядок можна подати у вигляді послідовності слів словника.

Приклад вихідного файлу

```
2
```

Задача "Абракадабра"

a	**a**	**b**	**r**	**a**	**k**
a	***b***	***r***	***a***	***k***	***a***
a	**k**	**a**	**a**	**b**	**r**
b	**r**	**a**	**k**	**a**	**a**
k	**a**	**a**	**b**	**r**	**a**
r	**a**	**k**	**a**	**a**	**b**

Під час своєї роботи алгоритм стискання даних методом «сортування блоку» застосовує до блоків даних перетворення, яке визначається наступним чином.

Рядок *P* називається ротацією рядка *S*, якщо він утворений циклічним зсувом символів *S*, тобто якщо $S=a_1a_2\ldots a_N$, де a_i — *i*–ий символ рядка *S*, то $P=a_pa_{p+1}\ldots a_Na_1\ldots a_{p-1}$, де $1\le p\le N$. Розглянемо таблицю *M* розміру $N\times N$, рядками якої є всі ротації рядка *S*, відсортовані у лексикографічному (словниковому) порядку за зростанням.

Нехай рядок *L* є останнім стовпчиком таблиці *M*. Пряме перетворення отримує на вхід рядок *S*, видає рядок *L* та число *K* —

номер рядка таблиці *M*, що містить рядок *S*. (Якщо таких рядків декілька, видається номер будь–якого з них).

Для *S*='abraka' таблицю *M* зображено на малюнку. Рядок *S* знаходиться у другому рядку таблиці *M*, *L*=‘karaab’.

Завдання

Напишіть програму ABRAKA, що виконує зворотнє перетворення, тобто отримує на вхід рядок *L* і число *K*, та видає рядок *S*.

Вхідні дані

Перший рядок вхідного файлу ABRAKA.DAT містить два цілих числа: *K* та *N*, $1 \le N \le 30000$, $1 \le K \le N$. Другий рядок містить *N* символів рядка *L* — маленьких латинських літер.

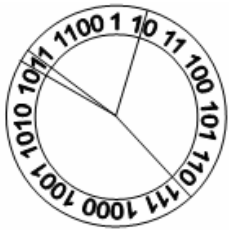
Вихідні дані

Єдиний рядок вихідного файлу ABRAKA.SOL повинен містити рядок *S*.

Приклад вхідних та вихідних даних

ABRAKA.DAT	ABRAKA.SOL
2 6 karaab	abraka

Задача "Циферблат"



На циферблаті записана послідовність чисел у двійковій системі числення. Циферблат може бути розбитий на сектори. Лінії розбиття можуть проходити як між числами, так і між цифрами одного числа, розбиваючи його на два чи більше чисел. Для кожного сектора можна порахувати суму чисел, які в ньому розташовані.

Кожне число в послідовності не дорівнює 0, та його запис починається з одиниці. Кількість цифр в двійковому запису числа не перевищує 25. Загальна кількість цифр на циферблаті не більша за 100.

На малюнку зображено звичний нам циферблат з числами від 1 до 12 (в дещо незвичному вигляді). Його розбито на 4 сектори. Суми для секторів будуть 1, 15, 18 та 36.

Завдання

Напишіть програму DIAL, що за заданою послідовністю визначає кількість різних розбиттів циферблату на сектори, такі що сума чисел у всіх секторах однакова.

Вхідні дані

В єдиному рядку вхідного файлу DIAL.DAT задана послідовність чисел. Числа послідовності розділені пропуском.

Вихідні дані

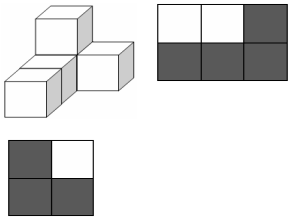
В єдиному рядку вихідного файлу DIAL.SOL повинно знаходитися натуральне число — кількість шуканих розбиттів циферблату на сектори.

Приклад вхідних та вихідних даних

DIAL.DAT	DIAL.SOL
101 1 1101	9

Задача "Кубики"

Тривимірна фігура складається з одиничних кубиків. За фігурою можна побудувати її фронтальну та праву проекції. Очевидно, що за цими двома проекціями не завжди можна відтворити фігуру.



Завдання

Напишіть програму CUBES, що отримує на вхід фронтальну та праву проекції фігури та визначає мінімальну та максимальну кількість кубиків, яку можна було б використати для побудови фігури із заданими проекціями.

Вхідні дані

В першому рядку вхідного файлу CUBES.DAT знаходиться три числа *N*, *M* та *K*, що задають розміри проекцій ($1 \le N, M, K \le 100$). Далі задаються дві проекції: спочатку фронтальна, а потім права. Проекція задається *N* рядками, кожний з яких складається з чисел 0 та 1, що розділені пропуском. Для фронтальної проекції таких чисел буде *M*, а для правої — *K*. 0 означає вільну клітину проекції, 1 — заповнену.

Вихідні дані

В єдиному рядку вихідного файлу CUBES.SOL повинно знаходитися два числа: мінімальна та максимальна кількість кубиків, які можна було б використати для побудови фігури із заданими проекціями.

Приклад вхідних та вихідних даних

CUBES.DAT	CUBES.SOL
2 2 3 1 0 1 1 0 0 1 1 1 1	4 7

II семестр (мова програмування Сі++)

Лабораторна робота №1 "Вступ у класи та об'єкти. Елементи об'єктного підходу: модульність та обмеження доступу"

Мета роботи: порівняння об'єктно-орієнтованого та функціонального підходів; початкове знайомство з класами, об'єктами та головними елементами об'єктного підходу.

Завдання: Створити клас для обробки записів бази даних у відповідності з наданим варіантом. Розмістити інтерфейс класу у заголовочному файлі, а визначення функцій та головну функцію програми – у двох окремих файлах. Передбачити можливість роботи з довільним числом записів, а також реалізувати окремими функціями класу:

- конструктори без параметрів та з параметрами ;
- додавання;
- знищення;
- виведення інформації на екран;
- пошук потрібної інформації за конкретною ознакою;
- редагування записів;
- сортування за різними полями.

Використайте захищення даних для ізоляції елементів-даних класу від підпрограм, в яких цей клас використовується. Програма повинна містити меню для перевірки всіх методів класу.

Примітка. Завдання необхідно розв'язати двома способами :

- з використанням функціонального підходу;
- з використанням об'єктно-орієнтованого підходу.

№	*Предметна область БД*	*Поля бази даних*
1.	„бібліотека"	Інвентарний номер, автор, назва, кількість сторінок, рік видання.
2.	„телефонний довідник"	Прізвище, ім'я, по батькові, домашня адреса, телефон.
3.	„розклад руху літаків"	Номер рейсу, тип літака, напрямок руху, періодичність вильоту.
4.	„колекція компакт-дисків"	Інвентарний номер, назва, об'єм диску, тип, дата запису.

5.	„записна книжка”	Прізвище, ім’я, по батькові, домашня адреса, телефон, електронна пошта.
6.	„предметний покажчик”	Слово; номера сторінок, де це слово зустрічається.
7.	„користувачі локальної мережі”	Прізвище, ім’я, по батькові, група, обліковий запис, тип облікового запису.
8.	„склад товарів”	Інвентарний номер, назва товару, вага, ціна, кількість.
9.	„рахунки банку”	Прізвище, ім’я, дата останньої операції, сума останньої операції, сума вкладу.
10.	„успішність студентів”	Прізвище, ім’я, номер групи, оцінки з трьох предметів.
11.	„камера схову ”	Прізвище, ім’я, дата здачі, термін зберігання, інвентарний номер та назва предмета.
12.	„каса продажу квитків”	Назва пункту, час відправлення, дата відправлення, час прибуття, дата прибуття, ціна квитка.
13.	„архів програм”	Назва програми, операційна система, розмір програми, дата запису.
14.	„список файлів”	Ім’я файла, розширення, розмір, дата створення, атрибути.
15.	“розклад пар”	Номер пари, предмет, прізвище викладача, форма заняття.

Лабораторна робота №2
“ Класова ієрархія та механізм успадкування. Віртуальність та поліморфізм."

Мета роботи: навчитися створювати ієрархії класів та використовувати віртуальність і поліморфізм .

Завдання: Створити клас для зберігання бази даних, вказаної у варіанті, із вказаними полями. Утворити похідний клас, залучивши до нього як мінімум два додаткових поля таким чином, щоб клас набув більшої спеціалізованості. Для другого класу використати конструктор, аби він містив усі аргументи, необхідні для ініціалізації

об’єкту похідного класу. Створіть необхідні функції, що дозволяють виводити інформацію на екран та можливість додавати та знищувати записи.

Варіант	*Предметна область БД*	*Поля БД*
1.	„предметний покажчик”	Слово; номера сторінок, де це слово зустрічається.
2.	„користувачі локальної мережі”	Прізвище, ім’я, по батькові, група, обліковий запис, тип облікового запису.
3.	„колекція компакт-дисків”	Назва, розмір диску, тип, дата запису.
4.	„перелік товарів”	Назва товару, вага, ціна, кількість.
5.	„домашня бібліотека”	Автор, назва, кількість сторінок, рік видання.
6.	„рахунки банку”	Прізвище, ім’я, дата останньої операції, сума вкладу.
7.	„успішність студентів”	Прізвище, ім’я, номер групи, оцінки з трьох предметів.
8.	„телефонний довідник”	Прізвище, ім’я, по батькові, домашня адреса, телефон.
9.	„студентський журнал”	Прізвище, ім’я, домашня адреса, телефон, дата народження.
10.	„список файлів”	Ім’я файла, розширення, розмір, дата створення, атрибути.
11.	“розклад пар на один день”	Номер пари, предмет, прізвище викладача, форма заняття
12.	„записна книжка”	Прізвище, ім’я, по батькові, домашня адреса, телефон, електронна пошта.
13.	„камера схову”	Прізвище, ім’я, дата здачі, термін зберігання, назва предмета.
14.	„каса продажу квитків”	Назва пункту, час відправлення, дата відправлення, час прибуття, дата прибуття, ціна квитка.
15.	„архів програм”	Назва програми, операційна система, розмір програми, примітка

II. Спроектуйте ієрархію класів для представлення графічних об'єктів. Головним базовим класом для усіх об'єктів є клас *Point* - точка на площині (у просторі) з її координатами. Опис класів слід розмістити у заголовочному файлі, а визначення функцій і головну функцію програми – в двох окремих файлах. Передбачте методи для створення об'єкта, його переміщення на екрані, зміни розмірів та кольору, обертання на заданий кут. Використайте захищення даних для ізоляції елементів-даних класу від підпрограм, в яких цей клас використовується, а також поліморфізм для визначення дії певних функцій у класовій ієрархії. Напишіть головну функцію, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

1. коло	16. трапеція
2. кільце	17. графічне вікно
3. паралелепіпед у просторі	18. еліпс
4. сфера	19. куб у просторі
5. прямокутник	20. куля
6. вектор на площині	21. курсор на екрані
7. багатокутник	22. ламана лінія
8. рівнобедрений трикутник	23. п'ятикутник
9. відрізок у просторі	24. паралелограм
10. відрізок на площині	25. ромб
11. шестикутник	26. сектор
12. вектор у просторі	27. тетраедр у просторі
13. курсор на екрані	28. трикутна призма у просторі
14. квадрат	29. прямокутний трикутник
15. конус	30. циліндр

Лабораторна робота №3
"Перевантаження операторів. Використання об'єктів потоків"

Мета роботи: навчитися реалізовувати перевантаження операторів та ознайомитися з використанням потоків.

Завдання: I. Описати клас, що реалізовує вказаний нижче тип даних. Клас повинен містити множину конструкторів для створення об'єктів певного типу (конструктор по замочуванню та з параметрами,

конструктор копії) та подані у таблиці операції над об'єктами класу (плюс обов'язково операції присвоювання та порівняння) з використанням механізму перевантаження операцій:

Варіант	*Тип даних*	*Операції*
1	"матриця"	Віднімання, множення, обчислення оберненої матриці
2	„комплексні числа"	сума, добуток, різниця, частка
3	„вектор у просторі".	додавання векторів, векторний добуток двох векторів
4	„множина"	вилучення елемента, об'єднання множин, перетин множин
5	"матриця"	додавання, частка, обчислення транспонованої матриці
6	„вектор у просторі"	віднімання та складання векторів, порівняння векторів
7	„множина"	додавання елемента, різниця множин, індексування
8	„дроби"	віднімання, множення
9	„множина"	додавання елемента, перетин множин, індексування
10	„рядок"	об'єднання рядків, копіювання рядків
11	„резервуар з водою"	змішування, переливання
12	„дроби"	додавання, ділення, інкремент, декремент
13	„цілі числа"	інкремент, декремент, додавання, віднімання, логічні операції
14	„вектор у площині"	додавання, множення вектора на число
15	„рядок"	Виокремлення підрядка за допомогою перевантаження операції ().

Написати програму, яка демонструє роботу з об'єктами цього класу. Програма повинна містити меню для перевірки усіх методів класу і операцій. Організувати виведення та введення даних за допомогою класів-потоків *cin* та *cout*.

II. Виконати завдання, подані в таблиці з використанням файлових потоків і методів обробки помилок.

Вхідні дані необхідно прочитати з файла *input.txt,* а всі результати роботи програми вивести на екран і записати у файл *output.txt*.

Вар	*Написати програму, яка ...*
1	Написати програму, яка копіює вміст вхідного файла у вихідний; інвертує рядок, подаючи його у зворотному вигляді; підраховує кількість чисел у тексті; видаляє всі слова, що починаються з голосних літер.
2	Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів у тексті, які закінчуються на голосну літеру; знаходить всі слова, довжина яких менша п’яти символів; видаляє всі слова, які містять хоча б одну латинську літеру
3	Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість великих літер у тексті; визначає слова, що мають найменшу кількість літер; видаляє всі слова, що починаються з малої літери
4	Написати програму, яка копіює вміст вхідного файла у вихідний; перевіряє, чи співпадає кількість відкритих і закритих дужок у введеному рядку (перевірити для круглих та квадратних дужок); знаходить найдовше слово; видаляє всі слова, що складаються тільки з латинських літер
5	Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів непарної довжини; замінює всі слова, записані кирилицею на аналогічні, записані латиницею (обробка – obrobka); видаляє всі слова, які починаються і закінчуються на голосну літеру
6	Написати програму, яка копіює вміст вхідного файла у вихідний файл; підраховує кількість слів у тексті, які починаються з голосної літери; знаходить всі слова, які містять непарну кількість приголосних літер; видаляє всі числа з тексту
7	Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів, які мають непарну довжину; виводить на екран частоту входження кожної літери у тексті; перевіряє правильність розташування круглих дужок у тексті; видаляє всі непарні числа з тексту.

Вар	*Написати програму, яка ...*
8	Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість слів, які містять однакову кількість голосних і приголосних літер; визначає найдовше слово; видаляє з тексту всі слова-паліндроми
9	Написати програму, яка копіює вміст вхідного файла у вихідний; підраховує кількість різних слів, що входять до заданого тексту; визначає кількість використаних символів; видаляє всі слова, що мають подвоєні літери
10	Написати програму, яка копіює вміст вхідного файла у вихідний файл; підраховує кількість чисел у тексті (не цифр, а саме чисел); виділяє всі слова, що складаються тільки з латинських літер; видаляє кожне друге слово
11	Написати програму, яка копіює вміст вхідного файла у вихідний; виводить на екран всі символи, які розташовані після першого символу „:”; підраховує кількість речень, що містять непарну кількість слів; видаляє з тексту всі слова, які розташовані після ком
12	Написати програму, яка копіює вміст вхідного файлу у вихідний; підраховує кількість цифр у тексті; визначає слова, що починаються з приголосних літер; знищує всі слова, які починаються і закінчуються за одну й ту ж літеру
13	Написати програму, яка копіює вміст вхідного файлу у вихідний; підраховує кількість розділових знаків у тексті; виводить всі слова, що мають парну кількість літер; міняє місцями першу і останню літери кожного слова
14	Написати програму, яка копіює вміст вхідного файлу у вихідний; підраховує кількість слів у тексті; виділяє слово, що містить найбільшу кількість голосних літер; видаляє з тексту всі непотрібні пробіли
15	Написати програму, яка копіює вміст вхідного файлу у вихідний; замінює всі великі літери, що входять до тексту на відповідні малі; визначає найдовше слово; видаляє всі слова, що містять непарну кількість приголосних літер

Лабораторна робота №4-5
“Шаблони функцій та шаблони класів. Параметризовані контейнерні класи.”

Мета роботи: ознайомитися із базовими механізмами використання шаблонів функцій та шаблонів класів, навчитися створювати та використовувати параметризовані функції та параметризовані контейнерні класи.

Завдання :

I. Контейнерний клас описує та забезпечує набір дій над даними параметризованого масиву, розмірність якого визначається під час роботи програми. Усі обчислення та перетворення повинні бути реалізовані у вигляді функцій-члені класу.

Варіант 1

В масиві обчислити:

- номер елемента масиву, найближчого до середньоарифметичного його значень;
- різниця елементів масиву, що розташовані між першим від'ємним та другим додатним елементами.

Перетворити масив таким чином, щоб в його першій половині розташовувались елементи, що стоять в парних позиціях, а в другій половині – елементи, що стоять в непарних позиціях.

Варіант 2

Дана прямокутна матриця. Визначити:

- кількість від’ємних елементів в тих рядках, які містять хоча б один нульовий елемент;
- суму модулів елементів, які розташовані після першого додатного елемента

Впорядкувати елементи матриці за спаданням модулів елементів

Варіант 3

У довільній матриці обчислити:

- кількість елементів масиву, рівних нулю;
- суму елементів масиву, які лежать в діапазоні від A до B.

Впорядкувати елементи масиву за спаданням модулів елементів

Варіант 4

В одновимірному масиві елементів, обчислити:

- номер максимального за модулем елемента;
- суму модулів елементів, які розташовані після першого додатного елемента.

Перетворити масив таким чином, щоб спочатку розташовувались всі елементи, ціла частина яких лежить в інтервалі [a,b], а потім – всі інші

Варіант 5

В масиві обчислити:

- мінімальний за модулем елемент масиву;
- суму модулів елементів, які розташовані після першого від'ємного елемента.

Стиснути масив, видаливши з нього всі елементи, величина яких знаходиться на інтервалі [a,b]. Місце, яке звільниться в кінці масиву заповнити символом чи числом з клавіатури.

Варіант 6

В масиві обчислити:

- мінімальний за модулем елемент масиву;
- суму модулів елементів масиву, розташованих після першого елемента, рівного нулю.

Перетворити масив таким чином, щоб в першій його половині розташовувались елементи, що стоять на парних позиціях, а в другій половині – елементи, що стоять в непарних позиціях.

Варіант 7

У матриці обчислити:

- максимальний за модулем елемент масиву;
- суму елементів масиву, що розташовані між першим і другим додатними елементами.

Перетворити матрицю таким чином, щоб всі елементи, рівні нулю, розташовувались після всіх інших.

Варіант 8

Дана прямокутна матриця. Визначити:

- кількість рядків, які не містять жодного нульового елемента;

- максимальне із чисел, що зустрічається в заданій матриці більше одного разу

Перетворити матрицю таким чином, щоб всі елементи, рівні нулю, розташовувались на початку всіх інших.

Варіант 9

Дана прямокутна матриця. Визначити:

- кількість стовпців, які не містять жодного нульового елемента.
- кількість елементів, менших за a , але більших b.

Переставляючи рядки заданої матриці, розташувати їх у відповідності із зростанням суми значень у стовпцях.

Варіант 10

В одномірному масиві обчислити:

- добуток елементів масиву з парними номерами;
- суму елементів масиву, які розташовані між першим і останнім нульовими елементами.

Впорядкувати масив таким чином, щоб спочатку розташовувались всі додатні елементи, а потім – всі від'ємні (елементи, рівні 0 вважати додатними).

Варіант 11

Дана прямокутна матриця. Визначити :

- кількість стовпців, які містять хоча б один нульовий елемент;
- номер рядка, в якому знаходиться найдовша серія з однакових елементів.

Впорядкувати масив таким чином, щоб спочатку розташовувались всі серії з однакових елементів, а потім – всі решта елементів.

Варіант 12

В одномірному масиві, що складається з N дійсних елементів, обчислити:

- суму елементів масиву з непарними елементами;
- суму елементів масиву, які розташовані між першим і останнім від'ємними елементами.

Перетворити масив, видаливши з нього всі елементи, модуль яких не перевищує число, що вводиться з клавіатури. Елементи, які звільняться в кінці масиву заповнити нулями.

Варіант 13

Дана прямокутна матриця. Визначити :

- добуток елементів в тих рядках, які не містять від'ємних елементів;
- максимум серед сум елементів діагоналей, паралельних головній діагоналі матриці.

Перетворити матрицю, видаливши з неї всі елементи, модуль яких не перевищує число, що вводиться з клавіатури. Елементи, які звільняться в кінці масиву, заповнити нулями.

Варіант 14

В одномірному масиві, що складається з N дійсних елементів, обчислити:

- максимальний елемент масиву;
- суму елементів масиву, що розташовані до останнього додатного елемента.

Перетворити масив, видаливши з нього всі елементи, модуль яких знаходиться в інтервалі [a,b]. Елементи, які звільняються в кінці масиву заповнити нулями.

Варіант 15

Дана прямокутна матриця. Визначити :

- суму елементів в тих стовпцях, які не містять від'ємних елементів;
- мінімум серед сум модулів елементів діагоналей, паралельних побічній діагоналі матриці.

Перетворити матрицю таким чином, щоб всі елементи, що дорівнюють символу, що вводиться з клавіатури, розташовувались на початку всіх інших.

II. Реалізувати контейнерний клас та необхідні функції-маніпулятори над його елементами.

Варіант 1

Описати параметризований клас стеку, що моделює роботу звичайного калькулятора з основними арифметичними діями. Для ілюстрації його роботи використайте постфіксну нотацію у формі "операнд-операнд-оператор". Протестуйте роботу даного калькулятору для різних типів операндів.

Варіант 2

Описати клас, що реалізує бінарне дерево, передбачити можливість додавання нових елементів, видалення існуючих, пошуку елемента за ключем, а також послідовного доступу до всіх елементів.

Написати програму, що використовує цей клас для представлення англо-російського словника. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу. Передбачити можливість формування словника з файлу і з клавіатури.

Варіант 3

Описати клас, що реалізує стек. Написати програму, що використовує цей клас для відшукання проходу по лабіринті.

Лабіринт представляється у виді матриці, що складає з квадратів. Кожен квадрат або відкритий, або закритий. Вхід у закритий квадрат заборонений. Якщо квадрат відкритий, то вхід у нього можливий з боку, але не з кута. Кожен квадрат визначається його координатами в матриці. Після відшукання проходу програма друкує знайдений шлях у виді координат квадратів.

Варіант 4

Описати клас „предметний покажчик”. Кожен компонент покажчика містить слово і номери сторінок, на яких це слово зустрічається. Кількість номерів сторінок, що відносяться до одного слова, від одного до десяти. Передбачити можливість формування покажчика з клавіатури і з файлу, висновку покажчика, висновку номерів сторінок для заданого слова, видалення елемента з покажчика.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 5

Описати параметризований клас стеку, що моделює роботу звичайного калькулятора з основними арифметичними діями. Для його ілюстрації його роботи використайте інфіксну нотацію у формі "операнд-оператор-операнд". Протестуйте роботу даного калькулятору для різних типів операндів.

Варіант 6

Описати клас "гаражна стоянка", що має одну лінію для стоянки автомашин. В'їзд та виїзд відбувається з різних кінців лінії. Моделювання виїзду машини з автопарку відбувається так, що техніка, яка заважає виїзду, вилучається, а потім, після виведення потрібного авто зі стоянки, повертається на місце у тому ж порядку слідування. Програма повинна виводити повідомлення про прибуття та виїзд будь-якої машини, видавати довідку про наявність конкретної машини в гаражі та відображати стан лінії для стоянки машин у поточний момент.

Варіант 7

Описати клас "черга у крамниці", що має одну лінію черги довжиною не більше певного значення, що ініціалізується у програмі. Вилучення та додавання відбувається за правилами лінійного списку "черга". Проведіть моделювання процесу черги, передбачивши такі події, як відкриття магазину, перерву на обід та відновлення роботи по його закінченню, кінець роботи та здача каси. Програма повинна виводити повідомлення про нового споживача, що стає у чергу, та про обслуговування особи, що залишає чергу, вказуючи грошовий вираз суми операції останнім. Потрібно постійно відображати стан черги у поточний момент.

Додаткове завдання. Реалізуйте успадкування класу, що реалізує пріоритетну чергу. При додаванні елемента у таку чергу порядковий номер елемента повинен визначатися його пріоритетом.

Варіант 8

Описати клас "гаражна стоянка", що має одну лінію для стоянки автомашин. В'їзд та виїзд відбувається з одного кінця лінії. Моделювання виїзду машини з автопарку відбувається так, що техніка, яка заважає виїзду, вилучається, а потім, після виведення потрібного авто зі стоянки, повертається на місце у тому ж порядку слідування. Програма повинна виводити повідомлення про прибуття та виїзд будь-якої машини, видавати довідку про наявність конкретної машини в гаражі та відображати стан лінії для стоянки машин у поточний момент.

Варіант 9

Описати клас, що реалізує стек. Написати програму, що використовує цей клас для моделювання Т-образного сортувального вузла на залізниці. Програма повинна розділяти на два напрямки склад, що складається з вагонів двох типів (на кожен напрямок формується склад з вагонів одного типу). Передбачити можливість формування складу з файлу і з клавіатури.

Варіант 10

Написати програму, що містить поточну інформацію про книги в бібліотеці. Зведення про книги містять: номер УДК; прізвище і ініціали автора; назва; рік видання; кількість екземплярів даної книги в бібліотеці. Програма повинна забезпечувати:

- початкове формування даних про всі книги в бібліотеці у виді списку;
- при взятті кожної книги вводиться номер УДК, і програма зменшує значення кількості книг на одиницю чи видає повідомлення про те, що необхідної книги в бібліотеці немає, чи необхідна книга знаходиться на руках;
- при поверненні кожної книги вводиться номер УДК, і програма збільшує значення кількості книг на одиницю;
- по запиту видаються зведення про наявність книг у бібліотеці.

Варіант 11

Текст допомоги для деякої програми організований як лінійний список. Кожен компонент тексту допомоги містить термін (слово) і текст, що містить пояснення до цього терміну. Кількість рядків тексту, що відносяться до одного терміна, коливається від однієї до п'яти. Скласти програму, що забезпечує:

- початкове формування тексту допомоги;
- виведення тексту допомоги;
- виведення тексту, що пояснює заданий термін.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 12

У файловій системі каталог файлів організований як лінійний список. Для кожного файлу в каталозі міститься наступна інформація:

ім'я файла; дата створення; кількість звертань до файла. Скласти програму, що забезпечує:

- початкове формування каталогу файлів;
- виведення каталогу файлів;
- видалення файлів, дата створення яких менше заданої;
- вибірку файла за найбільшою кількістю звертань.

Варіант 13

На міжміській телефонній станції картотека абонентів, що містить інформацію про телефони і їхніх власників, організована як бінарне дерево. Скласти програму, що:

- забезпечує початкове формування картотеки у вигляді бінарного дерева;
- робить виведення усієї картотеки;
- вводить номер телефону і час розмови;
- виводить повідомлення на оплату телефонної розмови.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 14

Текст допомоги для деякої програми організований як лінійний список. Кожен компонент тексту допомоги містить термін (слово) і текст, що містить пояснення до цього терміну. Кількість рядків тексту, що відносяться до одного терміна, коливається від однієї до п'яти. Скласти програму, що забезпечує:

- початкове формування тексту допомоги;
- виведення тексту допомоги;
- виведення тексту, що пояснює заданий термін.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 15

Англо-український словник побудований як бінарне дерево. Кожен компонент містить англійське слово, якому відповідає українське слово і лічильник кількості звертань до даного компонента. Первісне дерево формується відповідно до англійського алфавіту. У процесі експлуатації словника при кожному звертанні до компонента в лічильник звертань додаватиметься одиниця. Скласти програму, що:

- забезпечує початкове введення словника з конкретними значеннями лічильників звертань;
- формує нове представлення словника у вигляді бінарного дерева за наступним алгоритмом:

  а). у старому словнику шукається компонент із найбільшим значенням лічильника звертань;

  б). знайдений компонент заноситься в новий словник і видаляється зі старого;

  в). перехід до п. а) до вичерпання вихідного словника;
- робить виведення вихідного і нового словників.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

ЧАСТИНА 4. ДОДАТКИ

4.1 Вбудований відлагоджувач програм

Система *Borland C++ 3.1* має в інтегрованому середовищі програмування вбудований відлагоджувач текстів програм. Для того, щоб можна було використовувати вбудований відлагоджувач, потрібно, щоб були включені відповідні опції. А для того, щоб можна було б скористатися зовнішнім відлагоджувачем, наприклад, таким, як *Turbo Debugger*, необхідно включити відлагоджувальну інформацію у файл виконання. Опція, що керує включенням відлагоджувальної інформації, знаходиться в меню Options | Debugger | Source Debugging. По замовчуванню ця опція увімкнена.

Відлагоджувач текстів програм не потребує дизасемблювання, а автоматично включає оригінальний текст програми у виконуваний файл. Він зв'язує скомпільований об'єктний код з кожним рядком тексту програми.

Відлагоджувач надає користувачу багато можливостей. Можна керувати виконанням програми встановленням точок переривання. Можна виконувати програму покроково, спостерігаючи за зміною значень змінних і виразів.

Покрокове виконання програми – це процес виконання одного оператора (точніше, одного рядка програми) за один крок.

Для покрокового виконання програми з використанням вбудованого відлагоджувача системи *Borland C++ 3.1* необхідно натиснути клавішу *F7*. Помітимо, що рядок програми, що містить *main*() у вікні редагування, буде виконаний. Це початок виконання програми. Помітимо також і те, що рядки з #include та прототипами функцій будуть пропущені, так як директиви препроцесора і оголошення прототипів функцій не генерують коду і автоматично пропускаються відлагоджувачем. Це саме стосується і оголошень змінних. Натискання клавіші *F7* еквівалентне вибору пункту меню Run | Trace into. Якщо натискати *F7* декілька разів, виділення буде переміщуватися від рядка до рядка. Виділяється той рядок, який буде виконуватися на наступному кроці.

Коли в програмі зустрічається виклик функції, виділений рядок переміщується в тіло цієї функції. Можна також виконувати програму без входження у функції. Це здійснюється натисканням клавіші *F8* (Run | Step over). Натискання *F7* і *F8* можна комбінувати в будь-якому порядку.

Встановлення точок переривання. При всіх перевагах покрокового виконання програми це може бути достатньо довгим процесом у великій програмі або при наявності циклів, особливо якщо місце програми, яке потрібно відлагодити, знаходиться надто далеко від початку програми. У відлагоджувачі передбачена можливість руху по програмі великими кроками. Перша можливість – виконання програми до рядка, в якому знаходиться курсор. Цю можливість можна реалізувати, встановивши курсор в потрібний рядок і натиснувши *F4* або вибравши пункт меню Run | Go to cursor.

Друга можливість – встановити точку переривання (*breakpoint*). Щоб встановити точку переривання, необхідно перемістити курсор у вікні редагування у той рядок, в якому необхідно призупинити виконання програми. Потім необхідно вибрати пункт меню Debug | Toggle breakpoint. Це також можна зробити за допомогою комбінації клавіш *Ctrl-F8*. Рядок буде виділений яскравим кольором (червоним). Повторний вибір пункту меню відмінить точку переривання, в якій знаходиться курсор.

В програмі можна встановлювати декілька точок переривання.

Якщо встановлена одна або декілька точок переривання, то можна почати виконання програми (Run | Run або *Ctrl-F9*). Виконання програми зупиниться на першій точці переривання, яка зустрінеться. Оператори в рядку, в якій знаходиться точка переривання, виконані не будуть. Продовжити виконання програми можна в покроковому режимі або знову натиснувши *Ctrl-F9* або *F4*. В будь-якому випадку при досягненні чергової точки переривання виконання програми буде призупинене.

При встановлених точках переривання можна вивести на екран список точок переривання, який дозволяє додавати (видаляти) точки переривання, встановлювати умови або лічильник зупинки

програми при досягненні точки переривання. При виборі Debug | Breakpoints … на екрані з'явиться відповідне діалогове вікно, вибравши в якому кнопку Edit отримаємо ще одне вікно, в якому можна модифікувати умови, і лічильник (count).

Встановлення програми на початок (Program Reset). Якщо виконання програми в подальшому непотрібне (знайдена помилка або потрібно повернутися на початок програми), потрібно вибрати пункт меню Run | Progran Reset або натиснути відповідну комбінацію клавіш *Ctrl-F2*. Сеанс відлагодження буде припинений, і програма буде готова до виконання з самого початку.

Спостереження за змінними. Дуже важлива властивість відлагодження – можливість переглядати поточне значення однієї або кількох змінних в процесі виконання програми.

Визначити змінні, значення яких необхідно проконтролювати, необхідно вибрати пункт меню Debug | Watches і вибрати Add watch. У відкритому вікні діалогу необхідно ввести ім'я змінної або вираз. Відлагоджувач відкриє вікно Watch, в якому буде змінна або вираз і значення змінної або виразу. Можна продовжити додавання змінних у вікно Watch.

В процесі виконання програми у відлагоджувальному (покроковому) режимі значення у вікні Watch будуть автоматично змінюватися. Якщо змінна глобальна, її значення доступне в будь-якому місці програми. Якщо ж змінна локальна, то її значення доступне лише в області видимості змінної. Якщо змінна недоступна, то у вікні Watch замість значення виведеться відповідне повідомлення.

При перегляді виразів у вікні Watch є два обмеження. По-перше, у виразі заборонений виклик функцій. По-друге, у виразі не можуть використовуватися макроси, визначені з використанням #define.

Відлагоджувач *Borland C++* дозволяє здійснювати форматоване виведення значень, які спостерігаються. Для встановлення формату використовується наступна форма :

expression, format_code.

Список кодів формату задано в таблиці 4.1:

У форматі F можна вказувати число значущих цифр після коми:

```
average, F5
```

Якщо формат не вказано, відлагоджувач сам підбирає відповідний тип вибору формату. Якщо змінна типу char *, відлагоджувач видасть не покажчик, а відповідний цьому покажчику рядок символів. Відлагоджувач дозволяє спостерігати і об'єкт мови Сі++, при цьому можна використовувати формат R.

Таблиця 4.1. Формати відображення змінних

Код формату	*Значення*
C	У вигляді символу
D	Десяткове число
F(n)	Число з плаваючою комою
H або X	Шістнадцяткове число
M	Показати пам'ять (dump)
P	Покажчик
R	Структури : вивести імена і значення членів
S	Вивести керуючі символи

Стек виклику функцій (Call stack). В процесі виконання програми можна викликати на екран стан стеку викликів функцій, використовуючи пункт меню Debug | Call stack. При цьому видається на екран послідовність вкладених викликів функцій зі значення фактичних параметрів. Локальні змінні і адреси повернення не видаються. Зручність цієї опції можна оцінити при відлагодженні рекурсивних функцій.

Обчислення і зміна значень. При виборі пункту меню Debug | Evaluate/Modify на екрані з'являється вікно діалогу, в якому можна задавати вираз, що не містить викликів функцій і макросів. Значення цього виразу висвітиться в другому рядку. Якщо вираз є величиною типу lvalue (наприклад, проста змінна), можна в нижньому рядку введення задати нове значення, натиснути кнопку Modify і продовжити виконання програми з новим значенням виразу або змінної.

Вікно Inspect. Хоча інформація про змінні при спостереженні у вікні Watch буває достатньо, можна отримати більш детальну інформацію, відкривши вікно Inspect. При вибору пункту меню Debug | Inspect відкриється вікно діалогу, в якому можна ввести ім'я змінної, і на екрані з'явиться більш детальна інформація, включаючи тип, адресу розміщення змінної в пам'яті та її значення. Особливо інформативне вікно Inspect при аналізі об'єктів типу клас. При перегляді класу можна встановити курсор всередині вікна Inspect на член цього класу (змінну або функцію), натиснути Enter і на екрані з'явиться ще одне вікно Inspect для цього члена класу. При покроковому виконанні програми дані у вікні Inspect автоматично змінюються. На відміну від вікна Watch, неможливо відкрити вікно Inspect для змінної ззовні області дії цієї змінної.

Можна відкрити декілька вікон Inspect, розмістивши їх в різних місцях екрану.

Регістри процесора. В процесі відлагодження можна переглядати вміст регістрів процесора і встановлених прапорців. При виборі Window | Registers на екрані з'явиться вікно з іменами і вмістом регістрів процесора.

Вікно Output. При видачі результатів в процесі відлагодження відображення проходить на користувацький екран (User Screen), який звичайно закритий інтегрованим середовищем. Щоб не переключатися під час відлагодження для перегляду виведення на екран, можна відкрити вікно Output, вибравши пункт меню Window | Output. Вікно, що відкрилося можна переміщувати по екрану і змінювати його розміри. Воно відображує ту частину User Screen, в якій пройшла остання зміна. При покроковому виконанні програми виведення на екран відображається в цьому вікні. Правда, є обмеження : у вікні відображається текстова інформація. Графічний режим у вікні Output не підтримується.

4.2 Таблиця символів ASCII

В ОС ДОС для позначення символів використовується американський національний стандартний код для обміну інформацією ASCII (American Standard Code for Information Interchange). Відповідно до нього код символу зберігається в одному байті, тому коди символів можуть приймати значення від 0 до 255. Всього існує 256 символів (таблиця 4.2)

Таблиця 4.2. Таблиця символів ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	☺ 1	☻ 2	♥ 3	♦ 4	♣ 5	♠ 6	• 7	◘ 8	○ 9	◙ 10	♂ 11	♀ 12	♪ 13	♫ 14	☼ 15
1	► 16	◄ 17	↕ 18	‼ 19	¶ 20	§ 21	▬ 22	↨ 23	↑ 24	↓ 25	→ 26	← 27	∟ 28	↔ 29	▲ 30	▼ 31
2	32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	− 45	. 46	/ 47
3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	95
6	` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	⌂ 127
8	А 128	Б 129	В 130	Г 131	Д 132	Е 133	Ж 134	З 135	И 136	Й 137	К 138	Л 139	М 140	Н 141	О 142	П 143
9	Р 144	С 145	Т 146	У 147	Ф 148	Х 149	Ц 150	Ч 151	Ш 152	Щ 153	Ъ 154	Ы 155	Ь 156	Э 157	Ю 158	Я 159
A	а 160	б 161	в 162	г 163	д 164	е 165	ж 166	з 167	и 168	й 169	к 170	л 171	м 172	н 173	о 174	п 175
B	░ 176	▒ 177	▓ 178	│ 179	┤ 180	╡ 181	╢ 182	╖ 183	╕ 184	╣ 185	║ 186	╗ 187	╝ 188	╜ 189	╛ 190	┐ 191
C	└ 192	┴ 193	┬ 194	├ 195	─ 196	┼ 197	╞ 198	╟ 199	╚ 200	╔ 201	╩ 202	╦ 203	╠ 204	═ 205	╬ 206	╧ 207
D	╨ 208	╤ 209	╥ 210	╙ 211	╘ 212	╒ 213	╓ 214	╫ 215	╪ 216	┘ 217	┌ 218	█ 219	▄ 220	▌ 221	▐ 222	▀ 223
E	р 224	с 225	т 226	у 227	ф 228	х 229	ц 230	ч 231	ш 232	щ 233	ъ 234	ы 235	ь 236	э 237	ю 238	я 239
F	Ё 240	ё 241	Є 242	є 243	Ї 244	ї 245	÷ 246	≈ 247	° 248	∙ 249	· 250	√ 251	n 252	2 253	■ 254	255

4.3 Розширені коди клавіатури

Як відомо, функція *getch()* повертає код натиснутої клавіші. Нульове значення сигналізує про натискання спеціальної клавіші або комбінації спеціальних клавіш. Отже, якщо *getch()* повертає нуль, то в такому разі при наступному звертанні до функції *getch()* за кодом можна визначити, яка саме клавіша була натиснутою.

Таблиця 4.3. Значення допоміжного байта для функціональних клавіш

		SHIFT	**CTRL**	**ALT**
F1	59	84	94	104
F2	60	85	95	105
F3	61	86	96	106
F4	62	87	97	107
F5	63	88	98	108
F6	64	89	99	109
F7	65	90	100	110
F8	66	91	101	111
F9	67	92	102	112
F10	68	93	103	113
F11	133	135	137	139
F12	134	136	138	140

Таблиця 4.4. Значення допоміжного байта для комбінацій клавіш з ALT

ALT-1	120	**ALT-A**	30	**ALT-K**	37	**ALT-U**	22
ALT-2	121	**ALT-B**	48	**ALT-L**	38	**ALT-V**	47
ALT-3	122	**ALT-C**	46	**ALT-M**	50	**ALT-W**	17
ALT-4	123	**ALT-D**	32	**ALT-N**	49	**ALT-X**	45
ALT-5	124	**ALT-E**	18	**ALT-O**	24	**ALT-Y**	21
ALT-6	125	**ALT-F**	33	**ALT-P**	25	**ALT-Z**	44
ALT-7	126	**ALT-G**	34	**ALT-Q**	16	**ALT-мінус**	74
ALT-8	127	**ALT-H**	35	**ALT-R**	19	**ALT-плюс**	78
ALT-9	128	**ALT-I**	23	**ALT-S**	31	**ALT - ***	55
ALT-0	129	**ALT-J**	36	**ALT-T**	20	**ALT - =**	131

Таблиця 4.5. Значення допоміжного байта для інших комбінацій клавіш

ALT-	43	←	75
Insert	82	↑	72
Home	71	→	77
PgUp	73	↓	80
PgDn	81	**CTRL -** ←	115
End	79	**CTRL -** →	116
Delete	83	**CTRL-END**	117
5 (цифрова)	76	**CTRL-Home**	119
Shift-Tab	15	**CTRL-PgDn**	118

Приклад реалізації обробки відслідковування натискань спеціальних клавіш.

```
char c=getch();
if (c==0)
{
   c=getch();
   switch(c)
   {
    case 75  :/* натиснута стрілка вліво */ break;
    case 77  :/* натиснута стрілка вправо */ break;
    case 72  :/* натиснута стрілка вгору */ break;
    case 80  :/* натиснута стрілка вниз */ break;
    case 103 :/* натиснута комбінація CTRL-F10 */
              break;
    case 46  :/* натиснута комбінація ALT-C*/ break;
   }
}
```

4.4 Функції стандартної бібліотеки

Функції для роботи із символами

Таблиця 4.6. Функції для роботи із символами

Функція	*Опис*	*Тип повернення*
isalnum(int c);	Перевірка, чи є символ літерою або цифрою.	int
isalpha(int c);	Перевірка, чи є символ літерою.	int
iscntrl(int c);	Перевірка, чи є символ керуючим.	int
isdigit(int c);	Перевірка, чи є символ десятковою цифрою.	int
isgraph(int c);	Перевірка, чи є символ видимим.	int

islower(int c);	Перевірка, чи є символ літерою нижнього регістру.	int
ispunct(int c);	Перевірка, чи є символ знаком пунктуації.	int
isspace(int c);	Перевірка, чи є символ пробільним.	int
isupper(int c);	Перевірка, чи є символ літерою верхнього регістру.	int
isxdigit(int c);	Перевірка, чи є символ шістнадцятковою цифрою.	int
tolower(int c);	Перетворення символу в нижній регістр.	int
toupper(int c);	Перетворення символу у верхній регістр.	int

Функції для роботи з каталогами (dir.h)

Таблиця 4.7. Функції для роботи з каталогами (dir.h)

chdir(char *pathname);	Зміна поточного робочого каталогу.	int
findfirst(char *pathname, struct ffblk *buf, int attr);	Початок пошуку файла або каталогу.	int
fnmerge(char *path, char *drive, char *dir, char *name,char *ext)	Складання імені файла із окремих частин.	void
fnsplit(char *path, char *drive, char *dir, char *name,char *ext);	Розкладання імені файла на окремі компоненти.	int
getcurdir(int drive,char *directory);	Повертає поточний каталог на вказаному диску.	int
getcwd(char *buf, int n);	Повертає повне ім'я поточного каталогу.	char *
getdisk(void);	Повертає поточний диск.	int
mkdir(char *pathname);	Створення нового каталогу.	int
mktemp(char *template);	Генерує унікальне ім'я файла.	char *
rmdir(const char *path);	Знищення каталогу.	int
searchpath(char *filename);	Продовження пошуку файла, початого функцією findfirst.	char *
setdisk(int drive);	Встановлення поточного диску.	int

Функції для роботи з ОС (dos.h)

Таблиця 4.8. Функції для роботи з ОС (dos.h)

absread(int drive, int nsect, int sectno, void *buffer);	Читання інформації із сектора.	int
abswrite(int drive, int nsect, int sectno, void *buffer);	Запис інформації у сектор.	int
bdos(int dosfun, unsigned dosdx, unsigned dosal);	Виклик MS-DOS.	int
ctrlbrk(int (*handler)(void));	Встановлення реакції на CTRL-Break.	void
delay(unsigned milliseconds);	Призупинення роботи програми на вказане число мілісекунд	void

getcbkr(void);	Повертає поточну встановлену реакцію на CTRL-Break.	int
getdate(struct date *datep);	Повертає поточну дату.	void
getdfree(int drive, struct dfree *dtable);	Повертає об'єм вільного місця на диску.	void
getfat(int drive, struct fatinfo *fatblkp);	Отримати інформацію FAT.	void
getfatd(struct fatinfo *dtable);	Отримати інформацію FAT про поточний диск.	void
getftime(int handle,struct ftime, *ftimep);	Повертає дату і час створення файла.	int
gettime(struct time *timep);	Повертає поточний системний час.	void
inp(unsigned portid);	Читає один байт з вхідного порта port.	int
inport(int portid);	Читає слово(два байти) із вхідного порта.	int
inportb(int portid);	Читає байт з порта введення.	unsigned char
int86x(int intno, union REGS *inregs, union REGS *outregs, struct SREGS *segregs);	Виконує системне переривання.	int
intr(int intno, struct REGPACK *preg);	Виконує системне переривання.	void
keep(unsigned char status, unsigned size);	Завершити роботу і залишити програму резидентною.	void
nosound(void);	Відключити звук.	void
outp(unsigned portid, int value);	Записати байт в порт.	int
peek(unsigned segment, unsigned offset); peekb(unsigned segment, unsigned offset);	Отримати значення байта або слова за адресою.	int char
poke(unsigned segment, unsigned offset, int value); pokeb(unsigned segment, unsigned offset, char value);	Записати значення байта або слова за адресою.	void void
settime(struct time *timep);	Встановити поточний час.	void
sleep(unsigned seconds);	Призупинити виконання програми на задану кількість секунд.	void
sound(unsigned frequency);	Генерувати звуковий сигнал із заданою частотою.	void

Функції для роботи з графічним режимом (graphics.h)

Таблиця 4.9. Функції для роботи з графічним режимом (graphics.h)

bar(int left, int top, int right, int bottom);	Малює зафарбований прямокутник.	void far
arc(int x, int y, int stangle, int endangle, int radius);	Малює дугу.	void far
bar3d(int left, int top, int right, int bottom, int depth, int topflag);	Вимальовує трьохвимірний стовпець.	void far
circle(int x, int y, int radius);	Малює коло.	void far
cleardevice(void);	Очищає екран.	void far
clearviewport(void);	Очищає графічне вікно.	void far
closegraph(void);	Закриває графічний режим.	void far
detectgraph(int far *graphdriver, int far *graphmode);	Повертає тип графічного драйвера.	void far
drawpoly(int numpoints, int far *polypoints);	Вимальовує ламану лінію.	void far
ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);	Малює еліптичну дугу від початкового кута до кінцевого.	void far
fillellipse(int x, int y, int xradius, int yradius);	Малює заштрихований еліпс.	void far
fillpoly(int numpoints, int far *polypoints);	Малює і штрихує багатокутник.	void far
floodfill(int x, int y, int border);	Заштриховує замкнену область.	void far
getaspectratio(int far *xasp, int far *yasp);	Повертає відношення сторін графічного екрану.	void far
getbkcolor(void);	Повертає поточний колір фону.	int far
getcolor(void);	Повертає поточний колір.	int far
getfillpattern(char far *pattern);	Повертає поточний тип штриховки.	void far
getfillsettings (struct fillsettingstype far *fillinfo);	Повертає поточний тип і колір штриховки.	void far
getimage(int left, int top, int right, int bottom, void far *bitmap);	Зберегти бітовий образ частини екрана.	void far
getlinesettings(struct linesettingstype far *lineinfo);	Повертає поточний стиль, шаблон і товщину штриховки.	void far

getmaxcolor(void);	Повертає максимальний колір, який можна задавати в параметрах.	int far
getmaxx(void); getmaxy(void);	Повертають відповідно максимальну X-координату та Y-координати екрана.	int far int far
getpixel(int x, int y);	Повертає колір пікселя з координатами (x,y)	unsigned far
gettextsettings(struct textsettingstype far *texttypeinfo);	Повертає поточний шрифт, розмір та вирівнювання тексту.	void far
getx(void); gety(void);	Повертають відповідно X- та Y-координати поточного вказівника.	int far int far
graphresult(void);	Повертає код помилки для останньої графічної операції.	int far
imagesize(int left, int top, int right, int bottom);	Повертає число байт, що необхідні для зберігання прямокутної частини екрана.	unsigned far
initgraph(int far *graphdriver, int far *graphmode, char far *pathtodriver);	Ініціалізація графічного режиму роботи адаптера.	void far
line(int x1, int y1, int x2, int y2);	Малює лінію від точки (x1,y1) до точки (x2,y2).	void far
linerel(int dx, int dy);	Малює лінію від поточного положення вказівника до точки, заданої приростом координат.	void far
lineto(int x, int y);	Малює лінію від поточного положення вказівника до заданої точки.	void far
moverel(int dx, int dy);	Переміщує вказівник до точки, заданої приростом координат.	void far
moveto(int x, int y);	Переміщує вказівник до точки з заданими координатами.	void far
outtext(char far *textstring);	Виводить текстовий рядок на екран.	void far
outtextxy(int x, int y, char far *textstring);	Виводить текстовий рядок в задане місце екрана.	void far
pieslice(int x, int y, int stangle, int endangle, int radius);	Малює і штрихує сектор кола.	void far

putimage(int left, int top, void far *bitmap, int op);	Виводить бітовий образ на екран.	void far
putpixel(int x, int y, int color);	Виводить точку з заданими координатами і кольором.	void far
rectangle(int left, int top, int right, int bottom);	Малює прямокутник.	void far
sector(int x, int y, int stangle, int endangle, int xradius, int yradius);	Штрихує сектор еліпса.	void far
setaspectratio(int xasp, int yasp);	Змінює масштабний коефіцієнт відношення сторін екрана.	void far
setbkcolor(int color);	Встановлює колір фону.	void far
setcolor(int color);	Встановлює поточний колір	void far
setfillpattern(char far *upattern, int color);	Встановлює тип штриховки (довільний).	void far
setfillstyle(int pattern, int color);	Встановлює тип і колір штриховки.	void far
setlinestyle(int linestyle, unsigned upattern, int thickness);	Встановлює товщину і стиль лінії.	void far
settextjustify(int horiz, int vert);	Встановлює вирівнювання тектсу.	void far
settextstyle(int font, int direction, int charsize);	Встановлює поточний шрифт, стиль і розмір тексту.	void far
setviewport(int left, int top, int right, int bottom, int clip);	Визначає вікно для графічного виводу.	void far
textheight(char far *textstring);	Повертає висоту рядка у пікселях.	int far
textwidth(char far *textstring);	Повертає довжину рядка у пікселях.	int far

Математичні функції (math.h)

Таблиця 4.10. Математичні функції (math.h)

abs(int x);	Повертає модуль цілого числа.	int
acos(double x); acosl(long double (x));	Повертає арккосинус аргумента.	double long double
asin(double x); asinl(long double (x));	Повертає арксинус аргумента.	double long double

atan(double x); atanl(long double (x));	Повертає арктангенс аргумента.	double long double
atan2(double y, double x); atan2l(long double(y), long double (x));	Повертає арктангенс відношення аргументів.	double long double
ceil(double x); ceill(long double (x));	Заокруглення до найменшого цілого, більшого або рівного заданому числу.	double long double
cos(double x); cosl(long double x);	Обчислення косинуса.	double long double
cosh(double x); coshl(long double (x));	Обчислення гіперболічного косинуса.	double long double
exp(double x); exp(long double (x));	Повертає степінь числа e.	double long double
fabs(double x); fabsl(long double @E(x));	Повертає модуль числа (для дійсних чисел).	double long double
floor(double x); floorl(long double (x));	Заокруглення до найменшого цілого, меньшого або рівного заданому числу.	double long double
fmod(double x, double y); fmod(long double (x), long double (y));	Повертає залишок від ділення аргументів.	double long double
frexp(double x, int *exponent); frexp(long double (x), int *(exponent));	Виділяє з числа мантису і експоненціальну частину.	double long double
ldexp(double x, int expon); ldexpl(long double (x), int (expon));	Перетворює мантису і показник степеня в число.	double long double
log(double x); logl(long double (x));	Обчислює натуральний логарифм.	double long double
log10(double x); log10l(long double (x));	Обчислює десятковий логарифм.	double long double
modf(double x, double *ipart); modfl(long double (x), long double *(ipart));	Розбиває число на цілу і дробову частини.	double long double

pow(double x, double y); pow(long double (x), long double (y));	Підносить число до вказаного степеня.	double long double
sinl(long double x) sin(double x);	Обчислює синус аргумента.	long double double
sinh(double x); sinhl(long double (x));	Обчислює гіперболічний синус аргумента.	double long double
sqrt(double x); sqrtl(long double @E(x));	Обчислює квадратний корінь аргумена.	double long double
tan(double x); tanl(long double x);	Обчислює тангенс аргумента.	double long double
tanh(double x); tanhl(long double (x));	Обчислює гіерболічний тангенс аргумента.	double long double

Функції I/O (stdio.h)

Таблиця 4.11. Функції I/O (stdio.h)

clearerr(FILE *stream);	Очистка прапорця помилок для вказаного потоку.	void
fclose(FILE *stream);	Закриття потоку.	int
fcloseall(void);	Закриття всіх відкритих (на верхньому рівні) файлів (потоків).	int
feof(FILE *stream);	Перевірка на кінець потоку.	int
ferror(FILE *stream);	Перевірка прапорця помилок потоку.	int
fflush(FILE *stream);	Запис даних з буфера у потік.	int
fgetc(FILE *stream);	Читання символу з потоку.	int
fileno(FILE *stream);	Отримання дескриптора, зв'язаного з потоком.	int
fgetchar(void);	Читання символу із стандартного потоку введення	int
fgetpos(FILE *stream, fpos_t *pos);	Повертає поточну позицію у файлі.	int
fgets(char *s, int n, FILE *stream);	Читання рядка з потоку.	char *

fdopen(int handle, char *type);	Відкриття потоку (відкрити файл і зв'язати його з потоком).	FILE*
fprintf (FILE *stream, const char *format [, argument, ...]);	Запис даних в потік за форматом.	int
fputc(int c, FILE *stream);	Запис символу в потік.	int
fputchar(int c);	Запис символа в стандартний потік виведення.	int
fputs(const char *s, FILE *stream);	Запис рядка в потік.	int
fread(void *ptr, size_t size, size_t n, FILE *stream);	Читання даних з потоку.	size_t
freopen(const char *filename, const char *mode, FILE *stream);	Повторне відкриття потоку в новому режимі.	FILE *
fscanf (FILE *stream, const char *format [, address, ...]);	Читання даних з потоку за рядком формату.	int
fseek(FILE *stream, long offset, int whence);	Зміна позиції покажчика файла.	int
fsetpos(FILE *stream, const fpos_t *pos);	Переміщення вказівника файла відносно початку файла.	int
ftell(FILE *stream);	Повертає поточну позицію вказівника файла.	long
fwrite(const void *ptr, size_t size, size_t n, FILE*stream);	Запис даних із заданого буфера в потік.	size_t
getc(FILE *stream);	Читання символа з потоку.	int
getchar(void);	Читання символа з потоку stdin.	int
gets(char *s);	Читанн рядка із потоку stdin.	char*
getw(FILE *stream);	Читання слова (двох байт) із потоку.	int
printf (const char *format [, argument, ...]);	Запис даних в потік stdout за форматом.	int
putc(int c, FILE *stream);	Запис символа в потік.	int
putchar(int c);	Запис символа в потік stdout.	int
puts(const char *s);	Запис рядка в потік.	int
putw(int w, FILE *stream);	Запис слова (двох байт) в потік.	int
remove(const char *filename);	Знищення файла.	int
rename(const char *oldname, const char *newname);	Переіменування файла.	int
rewind(FILE *stream);	Встановлення вказівника файла на його початок.	void

scanf (const char *format [, address, ...]);	Читання даних з потоку stdin за форматом.	int
setbuf(FILE *stream, char *buf);	Встановлення буферизації потоку.	void
setvbuf(FILE *stream, char *buf, int type, size_t size);	Встановлення буферизації і розміру потока.	int
sprintf (char *buffer, const char *format [, argument, ...]);	Запис даних в рядок за форматом.	int
sscanf (const char *buffer, const char *format [, address, ...]);	Читання даних із рядка за форматом.	int
tempnam(char *dir, char *prefix);	Згенерувати ім'я тимчасового файла в заданому каталозі.	char *
ungetc(int c, FILE *stream);	Повертає символ в потік.	int
vfscanf(FILE *stream, const char *format, va_list arglist);	Читання даних з потоку з використанням списку аргументів.	int
vprintf (const char *format, va_list arglist);	Запис даних в стандартний потік виведення за форматом.	int
vsprintf(char *buffer, const char *format, va_list arglist);	Виведення рядка параметріву визначеному форматі.	int
vsscanf(const char *buffer, const char *format, va_list arglist);	Читає рядок, використовуючи список аргументів.	int

Функції для роботи з рядками (string.h)

Таблиця 4.12. Функції для роботи з рядками (string.h)

strcat(char *dest, const char *src);	Об'єднання рядків.	char *
strchr(const char *s, int c);	Пошук символа у рядку.	char *
strcmp(const char *s1, const char*s2);	Порівняння рядків.	int
strcpy(char *dest, const char *src);	Копіювання одного рядка в інший.	char *
strcspn(const char *s1, const char *s2);	Знайти перше входження символа із заданого набору символів в рядку.	size_t
strdup(const char *s);	Дублювання рядка.	char *
strerror(int errnum);	Повертає покажчик на рядок з описом помилки.	char *
strlen(const char *s);	Повертає довжину рядка.	size_t

strlwr(char *s);	Перетворити рядок у нижній регістр.	char *
strncat(char *dest, const char *src, size_t maxlen);	Об'єднує один рядок з n символами іншого.	char *
strncmp (const char *s1, const char *s2, size_t maxlen);	Порівнює один рядок з n символами іншого.	int
strncpy(char *dest, const char *src, size_t maxlen);	Копіює перші n символів одного рядка в інший.	char *
strnset(char *s, int ch, size_t n);	Заповнити n символів рядка в задане значення.	char *
strpbrk(const char *s1, const char *s2);	Знайти перше входження будь-якого символа із заданого набору в рядку	char *
strrchr(const char *s, int c);	Пошук першого входження заданого символа в рядку.	char *
strrev(char *s);	Інвертувати рядок.	char *
strncat(char *dest, const char *src, size_t maxlen);	Встановити всі символи рядка в задане значення.	char *
strspn(const char *s1, const char *s2);	Шакує перший символ одного рядка, відсутній в іншому.	size_t
strstr(const char *s1, const char *s2);	Шукає частину рядка в іншому рядку.	char *
strupr(char *s);	Перетворити рядок у верхній регістр.	char *

Консольні функції I/O (conio.h)

Таблиця 4.13. Консольні функції I/O (conio.h)

cgets(char *str);	Читання рядка з консолі.	char *
clreol(void);	Стирає частину рядка від поточного положення курсору до правої границі вікна.	void
clrscr(void);	Очищає екран або вікно.	void
cprintf (const char *format [arg, ...]);	Виведення рядка в текстове вікно за форматом.	int
cputs(const char *str);	Виведення рядка в текстове вікно.	int
cscanf (char *format [, address, ...]);	Читання даних з консолі з виконанням форматного перетворення.	int

delline(void);	Знищення поточного рядка в текстовому вікні.	void
getch(void);	Читання символа з консолі без ехо-друку.	int
getche(void);	Читання символа з консолі з ехо-друкум.	int
getpass(const char *prompt);	Читання 8 символів з консольного терміналу без ехо-дуку.	char *
gettext(int left, int top, int right, int bottom, void*destin);	Копіює частину текста з екрана в заданий буфер.	int
gettextinfo(struct text_info *r);	Дає інформацію про текстовий режим.	void
gotoxy(int x, int y);	Помістити курсор у вказане місце екрана або текстового вікна.	void
highvideo(void);	Встановлює високу яскравість символів.	void
insline(void);	Вставка порожнього рядка в текстове вікно.	void
inp(unsigned portid);	Читає байт з порта.	int
inport(int portid);	Читає 2 байта з порта.	int
inportb(int portid);	Читає байт з порта.	unsigned char
inpw(unsigned portid);	Читає з порта 2 байти.	unsigned
kbhit(void);	Перевірити натискання кнопки.	int
lowvideo(void);	Встановлює низьку яскравість символів.	void
movetext(int left, int top, int right, int bottom, int destleft, int desttop);	Копіює текст з однієї прямокутної частини екрана в іншу.	int
normvideo(void);	Встановлює стандартну яскравість символів.	void
outp(unsigned portid, int value);	Записати байт в порт.	int
outport(int portid, int value);	Записати 2 байти в порт.	void
outportb(int portid, unsigned char value);	Записати байт в порт.	void
outpw(unsigned portid, unsigned value);	Записати 2 байти в порт.	unsigned

putch(int ch);	Вивести символ в текстове вікно.	int
puttext(int left, int top, int right, int bottom, void*source);	Вивести текст із вказаного буферу на екран.	int
_setcursortype(int cur_t);	Встановити тип курсора.	void
textattr(int newattr);	Встановити колір тексту і тла.	void
textbackground(int newcolor);	Встановити колір тла тексту, що виводиться.	void
textcolor(int newcolor);	Встановити колір тексту, що виводиться.	void
textmode(int newmode);	Встановити текстовий режим.	void
ungetch(int ch);	Повертає символ, введений з клавіатури.	int
wherex(void); wherey(void);	Повертають відповідно поточну X або Y-координату.	int int
window(int left, int top, int right, int bottom);	Встановити координати поточного текстового вікна.	void

4.5 Пріоритети операцій

Таблиця 4.14. Пріоритети операцій

Операції (від вищого пріоритету до нижчого)	Порядок виконання
( ) [ ] –> . ::	→
! ~ ++ – – & * (тип) sizeof new delete	←
* / %	→
+ –	→
<< >>	→
< <= > >=	→
== !=	→
&	→
^	→
|	→
&&	→
||	→
?:	←
= += –= *= /= %= <<= >>= ^= |= &=	←
,	→

4.6 Основні комбінації клавіш середовища TC

Таблиця 4.15. Команди переміщення курсору

Операція	Комбінація клавіш
на символ вліво	CTRL – S або ←
на символ вправо	CTRL – D або →
на слово вліво	CTRL – A або CTRL - ←
на слово вправо	CTRL – F або CTRL - →
на рядок вгору	CTRL – E або ↑
на рядок вниз	CTRL – X або ↓
прокрутка вгору	CTRL – W
прокрутка вниз	CTRL – Z
на сторінку вгору	CTRL – R або PgUp
на сторінку вниз	CTRL – C або PgDn
попередня позиція курсору	CTRL – Q P

Таблиця 4.16. Команди вставки і вилучення

Операція	*Комбінація клавіш*
включити/виключити режим вставки	CTRL – V або Ins
вставити порожній рядок	CTRL – N
знищити рядок, на якому знаходиться курсор	CTRL – Y
знищити текст від поточної позиції курсору до кінця рядка	CTRL – Q Y
знищити символ, що знаходиться зліва	CTRL – H або Backspace
знищити символ, на якому знаходиться курсор	CTRL – G або Del
знищити слово, яке знаходиться справа	CTRL – T

Таблиця 4.17. Блочні команди

Операція	Комбінація клавіш
встановити початок блоку	CTRL – K B
встановити кінець блоку	CTRL – K K
виділити одне слово	CTRK – K T
копіювати блок	CTRK – K C
перемістити блок	CTRL – K V
знищити блок	CTRL – K Y
прочитати блок з диску	CTRL – K R
записати блок на диск	CTRL – K W
сховати/показати помітку	CTRL – K H
надрукувати блок	CTRL – K P
зсунути блок вліво	CTRL – K U
зсунути блок вправо	CTRL – K I

Таблиця 4.18. Керуючі команди

Операція	Комбінація клавіш
активізація верхнього меню	CTRL – K D або CTRL – K Q
збереження файла	CTRL – K S або F2
завантаження існуючого файла	F3
табуляція	CTRL – I або Tab
вибір режиму табуляції	CTRL – O T або CTRL – Q T
відновлення поточного рядка	CTRL– Q L
встановлення маркера позиції	CTRL – K (0, 1, 2, 3)
перейти на встановлений раніше маркер	CTRL – Q (0, 1, 2, 3)
префікс керуючого символу	CTRL – P
пошук	CTRL – Q F
пошук і заміна	CTRL – Q A
повторення останнього пошуку	CTRL – L

Таблиця 4.19. Режими пошуку

Опція пошуку	Значення
B	шукати назад
G	глобальний пошук
n	пошук *n* раз
N	заміна без запиту
U	ігнорування регістру
W	пошук тільки цілих слів
L	локальний пошук

Таблиця 4.20. Функціональні клавіші

Опція пошуку	Значення
F1	виклик контекстно-залежної допомоги
F2	збереження поточного файлу
F3	завантаження файла з диску
F4	виконати програму до поточної позиції курсору
F5	збільшити/зменшити вікно
F6	переключити активне вікно
F7	виконати один крок із входженням у функції
F8	виконати один крок без входження у функції
F9	створити програму (make)
F10	перехід між верхнім меню і редактором
ALT–F1	відобразити останню сторінку звертання до допомоги
ALT–F3	дозволяє вибрати один із файлів, які відкривалися раніше
ALT–F5	перейти до вікна результатів
ALT–F6	переключення між двома останніми робочими файлами
ALT–F7	перехід до попередньої помилки
ALT–F8	перехід до наступної помилки
ALT–F9	компілювати програму (*.OBJ)
ALT–B	перейти до меню Break/watch
ALT–C	перейти до меню Compile
ALT–D	перейти до меню Debug
ALT–E	перейти в редактор

ALT–F	перейти до меню File
ALT–O	перейти до меню Options
ALT–R	перейти до меню Run
ALT–X	вихід з ТС
Shift–F10	вивести інформацію про версію ТС
CTRL–F1	вивести допомогу про ідентифікатор, який визначається курсором
CTRL–F2	завершити покрокове виконання програми
CTRL–F3	активізувати вікно стека
CTRL–F4	обчислити вираз, показати значення змінної або змінити значення змінної
CTRL–F7	додати змінну у вікно спостереження
CTRL–F8	встановити контрольну точку
CTRL–F9	виконати програму

ЛІТЕРАТУРА

1. М.Уэйт, С.Прата, Д.Мартин „Язык Си”, Пер. с англ.-М.: Мир, 1988
2. Уинер Р. „Язык Турбо Си”, Пер с англ.-М.: Мир, 1991
3. Берри Р., Микинз Б. „Язык Си: введение для программистов”, Пер. с англ.-М.:Финансы и статистика, 1988
4. „Turbo C++”. Borland International. Inc. 1990.
5. Б. Страуструп „Введение в язык C++”, Киев, „Диа Софт”, 1995
6. Г. Буч. "Объектно-ориентированное проектирование с примерами применения", Киев, Диалектика, 1992.
7. Б. Страуструп „Язык программирования C++”; 2-е изд. : 1,2 т.т., Киев, „Диа Софт”, 1993
8. М.Эллис, Б.Страуструп „Справочное руководство по языку программирования C++ с комментариями”, М., Мир, 1992
9. С. Дьюхарст, Кети Старк „Программирование на C++”, Киев, „Диа Софт”, 1993
10. Т.Фэйсон „Объектно-ориентированное программирование на Borland C++ 4.5”, Четвертое издание, Киев, „Диалектика”, 1996.
11. И.Пол „Объектно-ориентированное программирование с использованием C++”, Киев, „Диа Софт”, 1995
12. Т.Сван „Освоение Borland C++ 4.5”, практический курс, 1 том, Киев, „Диалектика”, 1996
13. Т.Сван „Освоение Borland C++ 4.5”, Энциклопедия функций (2 том), Киев, „Диалектика”, 1996
14. Т.Сван „Освоение Borland C++ 5”, премьерное издание, Киев, “Диалектика”, 1996
15. Б. Страуструп „Язык программирования C++”, специальное издание, Москва, „Бином”, 2001
16. А. Архангельский „Программирование C++ Builder 5”, Москва, “Бином”, 2001
17. Т.А.Павловская „C/C++. Программирование на языке високого уровня”, „Питер”С-П, 2002
18. Т.Кормен, Ч.Лейзерсон, Р.Ривест „Алгоритмы: построение и анализ”,МЦНМО, М., 2000

ЗМІСТ

Для нотаток

Для нотаток

Войтенко Володимир Володимирович

Морозов Андрій Васильович

C/C++. Теорія та практика

Редактори : Войтенко В.В., Морозов А.В.
Комп'ютерний набір та верстка Войтенко В.В., Морозов А.В.

Підписано до друку 18.12.02 Формат 60x84 1/16 Друк офсетний.
Гарнітура Times New Roman. Ум. друк. арк. 20,25.
Тираж 400 екз. Зам. 79

Редакційно-видавничий відділ
Житомирського державного технологічного університету
10005, м. Житомир, вул. Черняхівського, 103.