Міністерство освіти і науки України

Донбаська державна машинобудівна академія

К О Н С П Е К Т Л Е К Ц І Й

з дисципліни " КОМП'ЮТЕРНІ ТЕХНОЛОГІЇ І ПРОГРАМУВАННЯ"

(Частина 1.Програмування мовою С)

для студентів спеціальності 151

Краматорськ 2018

Міністерство освіти і науки України

Донбаська державна машинобудівна академія

К О Н С П Е К Т Л Е К Ц І Й
з дисципліни "КОМП'ЮТЕРНІ ТЕХНОЛОГІЇ І ПРОГРАМУВАННЯ"
(Частина 1. Програмування мовою С)
для студентів спеціальності 151

Затверджено
на засіданні кафедри АПП
Протокол № від

Краматорськ 2018

УДК 621

Конспект лекцій з дисципліни "Комп'ютерні технології т програмирование" для студентів спеціальності 151/ Сост.: Е.В.Пищулина. - Краматорськ: ДГМА, 2018.- 144 с

Укладачі: Е.В.Пищулина, ст.преп.

Отв. за випуск Г. П. Клименко, проф.

ЗМІСТ

1 ОПИС МОВИ С

1.1 Елементи Мови С

1.1.1 Використовувані символи

Безліч символів використовуваних в мові СІ можна розділити на п'ять груп.

1. Символи, використовувані для творення ключових слів і ідентифікаторів (таблиця.1). До цієї групи входять прописні і рядкові букви англійського алфавіту, а також символ підкреслення. Слід зазначити, що однакові прописні і рядкові букви вважаються різними символами, оскільки мають різні коди.

Таблиця 1

Прописні букви латинського алфавіту	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Рядкові букви латинського алфавіту	a b c d e f g h i j k l m n o p q r s t u v w x y z
Символ підкреслення	_

2. Група прописних і рядкових букв російського алфавіту і арабські цифри (таблиця.2).

Таблиця 2

Прописні букви російського алфавіту	А Б В Г Д Е Ж З І ДО Л М Н ПРО П Р З Т У Ф Х Ц Ч Ш Щ Ы Ь Э Ю Я
Рядкові букви російського алфавіту	а б в г д е ж з і до л м н про п р з т у ф х ц ч ш щ ъ ы ь э ю я
Арабські цифри	0 1 2 3 4 5 6 7 8 9

3. Знаки нумерації і спеціальні символи (таблиця. 3). Ці символи використовуються з одного боку для організації процесу обчислень, а з іншої - для передачі компілятору певного набору інструкцій.

Таблиця 2

Символ	**Найменування**	**Символ**	**Найменування**
,	кома	)	кругла дужка права
.	точка	(	кругла дужка ліва

;	крапка з комою	}	фігурна дужка права
:	двокрапка	{	фігурна дужка ліва
?	знак питання	<	менше
'	апостроф	>	більше
!	знак оклику	[	квадратна дужка
|	вертикальна риса	]	квадратна дужка
/	дробова риса	#	номер
\	зворотна риса	%	відсоток
~	тильда	&	амперсанд
*	зірочка	^	логічне не
+	плюс	=	рівно
-	мину	"	лапки

4. Символи, що управляють і розділові. До тієї групи символів відносяться: пропуск, символи табуляції, перекладу рядка, повернення каретки, нова сторінка і новий рядок. Ці символи відділяють один від одного об'єкти, визначувані користувачем, до яких відносяться константи і ідентифікатори. Послідовність розділових символів розглядається компілятором як один символ (послідовність пропусків).

5. Окрім виділених груп символів в мові СІ широко використовуються так звані послідовності, що управляють, тобто спеціальні символьні комбінації, використовувані у функціях введення і виведення інформації. Послідовність, що управляє, будується на основі використання зворотної дробової риси (\) (обов'язковий перший символ) і комбінацією латинських букв і цифр (таблиця.4).

Послідовності виду \ddd і \xddd (тут d означає цифру) дозволяє представити символ з набору кодів ПЕВМ як послідовність вісімкових або шістнадцятиричних цифр відповідно. Наприклад символ повернення каретки може бути представлений різними способами:

\r - загальна послідовність, що управляє,

\015 - вісімкова послідовність, що управляє,

\x00D - шістнадцятирична послідовність, що управляє.

Слід зазначити, що в строкових константах завжди обов'язково задавати усі три цифри в послідовності, що управляє. Наприклад окрему послідовність \n (перехід на новий рядок), що управляє, можна представити як \010 або \xA, але в строкових константах необхідно задавати усі три цифри, інакше символ

або символи йдуть за послідовністю, що управляє, розглядатимуться як її бракуюча частина. Наприклад:

"ABCDE\x009FGH" ця строкова команда буде надрукована з використанням певних функцій мови CI, як два слова ABCDE FGH, розділені 8-у пропусками, в цьому випадку якщо вказати неповний рядок" ABCDE\, що управляє, x09FGH", то на друці з'явиться ABCDE=|=GH, оскільки компілятор сприйме послідовність \x09F як символ "=+=".

Відмітимо той факт, що, якщо зворотна дробова риса передує символу що не є послідовністю (тобто не включеному в таблицю.4), що управляє, і не є цифрою, то ця риса ігнорується, а сам символ представляється як літеральний. Наприклад:

символ \h представляється символом h в строковій або символьній константі.

Окрім визначення послідовності, що управляє, символ зворотної дробової риси (\) використовується також як символ продовження. Якщо за (\) слідує (\n), то обидва символи ігноруються, а наступний рядок є продовженням попередньої. Ця властивість може бути використана для запису довгих рядків.

1.1.2. Константи

Константами називаються перерахування величин в програмі. У мові CI розділяють чотири типи констант : цілі константи, константи з плаваючою комою, символьні константи і строковими літерали.

Ціла константа: це десяткове, вісімкове або шістнадцятиричне число, яке представляє цілу величину в одній з наступних форм : десятковою, вісімковою або шістнадцятиричною.

Десяткова константа складається з однієї або декількох десяткових цифр, причому перша цифра не має бути нулем (інакше число буде сприйнято як вісімкове).

Вісімкова константа складається з обов'язкового нуля і однієї або декількох вісімкових цифр (серед цифр мають бути відсутніми вісімка і дев'ятка, оскільки ці цифри не входять у вісімкову систему числення).

Шістнадцятирична константа розпочинається з обов'язкової послідовності 0x або 0X і містить одну або декілька шістнадцятиричних цифр (цифри є набором цифр шеснадцатеричной системи числення : 0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F)

Приклади цілих констант :

Десяткова константа	Вісімкова константа	Шістнадцятирична константа

16	020	0x10
127	0117	0x2B
240	0360	0XF0

Якщо вимагається сформувати негативну цілу константу, то використовують знак "-" перед записом константи (який називатиметься унарним мінусом). Наприклад: - 0x2A, - 088, - 16 .

Кожній цілій константі привласнюється тип, що визначає перетворення, які мають бути виконані, якщо константа використовується у виразах. Тип константи визначається таким чином:

- десяткові константи розглядаються як величини зі знаком, і їм привласнюється тип int (ціла) або long (довга ціла) відповідно до значення константи. Якщо константа менше 32768, то їй привласнюється тип int інакше long.

- вісімковим і шістнадцятиричним константам привласнюється тип int, unsigned int (беззнакова ціла), long або unsigned long залежно від значення константи згідно таблиці 5.

Для того, щоб будь-яку цілу константу визначити типом long, досить у кінці константи поставити букву "l" або "L". Приклад:

5l, 6l, 128L, 0105L, OX2A11L.

Константа з плаваючою точкою - десяткове число, представлене у вигляді дійсної величини з десятковою точкою або експонентою. Формат має вигляд:

[цифри].[цифри] [E|e [+|-] цифри] .

Число з плаваючою точкою складається з цілої і дробові частини і (чи) експоненти. Константи з плаваючою точкою представляють позитивні величини подвоєної точності (мають тип double). Для визначення негативної величини необхідно сформувати константне вираження, що складається зі знаку мінуса і позитивної константи.

Приклади: 115.75, 1.5E-2, - 0.025, .075, -0.85E2

Символьна константа - представляється символом ув'язненому в апострофи. Послідовність, що управляє, розглядається як поодинокий символ, допустимо її використати в символьних константах. Значенням символьної константи є числовий код символу. Приклади:

' '- пропуск,

'Q '- буква Q,

'\n' - символ нового рядка,

'\\' - зворотна дробова риса,

'\v' - вертикальна табуляція .

Символьні константи мають тип int і при перетворенні типів доповнюються знаком.

Строкова константа (літерал) - послідовність символів (включаючи строкові і прописні букви російського і латинського а також цифри) ув'язнені в лапки ("). Наприклад: "Школа N 35", "місто Тамбов", "YZPT КОД".

Відмітимо, що усі символи, що управляють, лапка ("), зворотна дробова риса (\) і символ нового рядка в строковому літералі і в символьній константі представляються відповідними послідовностями, що управляють. Кожна послідовність, що управляє, представляється як один символ. Наприклад, при друці літерала "Школа \n N 35" його частина "Школа" буде надрукована на одному рядку, а друга частина "N 35" на наступному рядку.

Символи строкового літерала зберігаються в області оперативної пам'яті. У кінець кожного строкового літерала компілятором додається нульовий символ, що представляється послідовністю \, що управляє, 0.

Строковий літерал має тип char[] . Це означає, що рядок розглядається як масив символів. Відмітимо важливу особливість, число елементів масиву дорівнює числу символів в рядку плюс 1, оскільки нульовий символ (символ кінця рядка) також є елементом масиву. Усі строкові літерали розглядаються компілятором як різні об'єкти. Строкові літерали можуть розташовуватися на декількох рядках. Такі літерали формуються на основі використання зворотної дробової риси і клавіші введення. Зворотна риса з символом нового рядка ігнорується компілятором, що призводить до того, що наступний рядок є продовженням попередньої. Наприклад: Строковый литерал имеет тип char[] . Это означает, что строка рассматривается как массив символов. Отметим важную особенность, число элементов массива равно числу символов в строке плюс 1, так как нулевой символ (символ конца строки) также является элементом массива. Все строковые литералы рассматриваются компилятором как различные объекты. Строковые литералы могут располагаться на нескольких строках. Такие литералы формируются на основе использования обратной дробной черты и клавиши ввод. Обратная черта с символом новой строки игнорируется компилятором, что приводит к тому, что следующая строка является продолжением предыдущей. Например:

"рядок невизначеної \n

довжини"

повністю ідентична літералу

"рядок невизначеною довгі" .

Для зчеплення строкових літералів можна використати символ (чи символи) пропуску. Якщо в програмі зустрічаються два або більше строкових літерала, розділених тільки пропусками, то вони розглядатимуться як один символьний рядок. Цей принцип можна використати для формування строкових літералів що займають більше за один рядок.

1.1.3. Ідентифікатор

Ідентифікатором називається послідовність цифр і букв, а також спеціальних символів, за умови, що першою коштує буква або спеціальний символ. Для утворення ідентифікаторів можуть бути використані рядкові або прописні букви латинського алфавіту. Як спеціальний символ може використовуватися символ підкреслення (_). Два ідентифікатори для утворення яких використовуються співпадаючі рядкові і прописні букви, вважаються різними. Наприклад: abc, ABC, A128B, a128b .

Важливою особливістю є те, що компілятор допускає будь-яку кількість символів в ідентифікаторі, хоча значимими є перші 31 символ. Ідентифікатор створюється на етапі оголошення змінної, функції, структури і тому подібне після цього його можна використати в подальших операторах програми, що розробляється. Слід зазначити важливі особливості при виборі ідентифікатора.

В перших, ідентифікатор не повинен співпадати з ключовими словами, із зарезервованими словами і іменами функцій бібліотеки компілятора мови СІ.

У других, слід звернути особливу увагу на використання символу (_) підкреслення в якості першого символу ідентифікатора, оскільки ідентифікатори побудовані таким чином, що, з одного боку, можуть співпадати з іменами системних функцій і (чи) змінних, а з іншого боку, при використанні таких ідентифікаторів програми можуть виявитися непереносними, тобто їх не можна використати на комп'ютерах інших типів.

У третіх, на ідентифікатори використовувані для визначення зовнішніх змінних, мають бути накладені обмеження, що формуються використовуваним редактором зв'язків (відмітимо, що використання різних версій редактора зв'язків, або різних редакторів накладає різні вимоги на імена зовнішніх змінних).

1.1.4. Ключові слова

Ключові слова - це зарезервовані ідентифікатори, які наділені певним сенсом. Їх можна використати тільки відповідно до значення відомим компілятору мови СІ.

Приведемо список ключових слів

```
auto    double   int  struct break  else  long  switch
register tupedef  char extern return void  case  float
unsigned default  for  signed union  do    if    sizeof
volatile continue enum short  while
```

Крім того в даній версії реалізації мови CI, зарезервованими словами являються:

_asm, fortran, near, far, cdecl, huge, paskal, interrupt .

Ключові слова far, huge, near дозволяють визначити розміри покажчиків на області пам'яті. Ключові слова _asm, cdelc, fortran, pascal служать для організації зв'язку з функціями написаними на інших мовах, а також для використання команд мови асемблера безпосередньо в тілі програми, що розробляється, на мові CI.

Ключові слова не можуть бути використані в якості ідентифікаторів.

1.1.5. Використання коментарів в тексті програми

Коментар - це набір символів, які ігноруються компілятором, на цей набір символів, проте, накладаються наступні обмеження. Усередині набору символів, який представляє коментар не може бути спеціальних символів тих, що визначають початок і кінець коментарів, відповідно (/* і */). Відмітимо, що коментарі можуть замінити як один рядок, так і декілька. Наприклад:

```
/* коментарі до програми */
/* початок алгоритму */
 чи
/* коментарі  можна  записати в наступному виді,  проте потрібно
     бути обережним, щоб усередині послідовності, яка ігнорується компілятором, не попалися оператори програми, які також ігноруватимуться
     */
```

Неправильне визначення коментарів.

```
/* коментарі до алгоритму /* рішення крайової задачі */ */
 чи
/* коментарі до алгоритму рішення */ крайового завдання */
типа
  описувач [=ініціатор] [,описувач [= ініціатор] ]...
```

1.2.1 Категорії типів даних

Ключові слова для визначення основних типів даних

Цілі типи:	Плаваючі типи:
char	float
int	double
short	long double
long	
signed	
unsigned	

Змінна будь-якого типу може бути оголошена як що не модифікується. Це досягається додаванням ключового слова const до специфікатору-типа. Об'єкти з типом const є даними використовувані тільки для читання, тобто цією змінною не може бути присвоєне нове значення. Відмітимо, що якщо після слова const відсутній специфікатор-типа, то мається на увазі специфікатор типу int. Якщо ключове слово const стоїть перед оголошенням складених типів (масив, структура, суміш, перерахування), то це призводить до того, що кожен елемент також має бути таким, що не модифікується, тобто значення йому може бути присвоєно тільки один раз. Переменная любого типа может быть объявлена как немодифицируемая. Это достигается добавлением ключевого слова const к спецификатору-типа. Объекты с типом const представляют собой данные используемые только для чтения, т.е. этой переменной не может быть присвоено новое значение. Отметим, что если после слова const отсутствует спецификатор-типа, то подразумевается спецификатор типа int. Если ключевое слово const стоит перед объявлением составных типов (массив, структура, смесь, перечисление), то это приводит к тому, что каждый элемент также должен являться немодифицируемым, т.е. значение ему может быть присвоено только один раз.

Приклади:

```
const double A=2.128E-2;
const B=286; (мається на увазі const int B=286)
```

Приклади оголошення складених даних будуть розглянуті нижче.

1.2.2. Цілий тип даних

Для визначення даних цілого типу використовуються різні ключові слова, які визначають діапазон значень і розмір області пам'яті, що виділяється під змінні (таблиця. 6).

Відмітимо, що ключові слова signed і unsigned необов'язкові. Вони вказують, як інтерпретується нульовий біт оголошуваної змінної, тобто, якщо вказано ключове слово unsigned, то нульовий біт інтерпретується як частину числа, інакше нульовий біт інтерпретується як

знаковий. У разі відсутності ключового слова unsigned ціла змінна вважається знаковою. У тому випадку, якщо специфікатор типу складається з ключового типу signed або unsigned і далі слідує ідентифікатор змінної, то вона розглядатиметься як змінна типу int. Наприклад: Отметим, что ключевые слова signed и unsigned необязательны. Они указывают, как интерпретируется нулевой бит объявляемой переменной, т.е., если указано ключевое слово unsigned, то нулевой бит интерпретируется как часть числа, в противном случае нулевой бит интерпретируется как знаковый. В случае отсутствия ключевого слова unsigned целая переменная считается знаковой. В том случае, если спецификатор типа состоит из ключевого типа signed или unsigned и далее следует идентификатор переменной, то она будет рассматриваться как переменная типа int. Например:

```
unsigned int n;
unsigned int b;
int c;        (мається  на увазі signed   int c  );
unsigned d;   (мається  на увазі unsigned int d  );
signed f;     (мається  на увазі signed   int f  ).
```

Відмітимо, що модификатор-типа char використовується для представлення символу (з масиву представлення символів) або для оголошення строкових літералів. Значенням об'єкту типу char є код (розміром 1 байт), що відповідає символу, що представляється. Для представлення символів російського алфавіту, модифікатор типу ідентифікатора даних має вигляд unsigned char, оскільки коди російських букв перевищують величину 127. Отметим, что модификатор-типа char используется для представления символа (из массива представление символов) или для объявления строковых литералов. Значением объекта типа char является код (размером 1 байт), соответствующий представляемому символу. Для представления символов русского алфавита, модификатор типа идентификатора данных имеет вид unsigned char, так как коды русских букв превышают величину 127.

Слід зробити наступне зауваження: в мові СІ не визначено представлення в пам'яті і діапазон значень для ідентифікаторів з модификаторами-типа int і unsigned int. Розмір пам'яті для змінної з модифікатором типу signed int визначається завдовжки машинного слова, яке має різний розмір на різних машинах. Так, на 16-ти розрядних машинах розмір слова рівний 2-м байтам, на 32-х розрядних машинах відповідно 4-м байтам, тобто тип int еквівалентний типам short int, або long int залежно від архітектури використовуваної ПЕВМ. Таким чином, одна і та ж програма може правильно працювати на одному комп'ютері і неправильно на іншому. Для визначення довжини пам'яті займаної змінної можна використати операцію sizeof мови СІ, що повертає значення довжини вказаного модифікатора-типу. Следует сделать следующее замечание: в языке СИ не определено представление в памяти и диапазон значений для идентификаторов с модификаторами-типа int и unsigned int. Размер памяти для переменной с модификатором типа signed int определяется длиной машинного слова, которое имеет различный размер на разных машинах. Так, на 16-ти разрядных машинах размер слова равен 2-м байтам, на 32-х разрядных машинах соответственно 4-м байтам, т.е. тип int эквивалентен типам short int, или long int в зависимости от архитектуры используемой ПЭВМ. Таким образом, одна и та же программа

может правильно работать на одном компьютере и неправильно на другом. Для определения длины памяти занимаемой переменной можно использовать операцию sizeof языка СИ, возвращающую значение длины указанного модификатора-типа.

Наприклад:

```
a = sizeof(int);
b = sizeof(long int);
c = sizeof(unsigned long);
d = sizeof(short);
```

Відмітимо також, що вісімкові і шістнадцятиричні константи також можуть мати модифікатор unsigned. Це досягається вказівкою префікса u або U після константи, константа без цього префікса вважається знаковою.

Наприклад:

```
0xA8C   (int  signed   );
01786l  (long signed   );
0xF7u   (int  unsigned );
```

1.2.3. Дані плаваючого типу

Для змінних, що представляють число з плаваючою точкою використовуються наступні модификаторы-типа: float, double, long double (у деяких реалізаціях мови long double СІ відсутнє).

Величина з модификатором-типа float займає 4 байти. З них 1 байт відводиться для знаку, 8 біт для надмірної експоненти і 23 біта для мантиси. Відмітимо, що старший біт мантиси завжди дорівнює 1, тому він не заповнюється, у зв'язку з цим діапазон значень змінної з плаваючою точкою приблизно рівний від 3.14E-38 до 3.14E+38.

Величина типу double займає 8 біт в пам'яті. Її формат аналогічний формату float. Біти пам'яті розподіляються таким чином: 1 біт для знаку, 11 біт для експоненти і 52 біта для мантиси. З урахуванням опущеного старшого біта мантиси діапазон значень рівний від 1.7E-308 до 1.7E+308.

Приклади:

```
float f, a, b;
double x, y;
```

1.2.4. Покажчики

Покажчик - це адреса пам'яті, що розподіляється для розміщення ідентифікатора (ідентифікатором може виступати ім'я змінної, масиву, структури, строкового літерала). У тому випадку, якщо змінна оголошена як покажчик, то вона містить адресу пам'яті, по якій може знаходиться скалярна величина будь-якого типу. При оголошенні змінної типу покажчик, необхідно визначити тип об'єкту даних, адреса яких міститиме змінна, і ім'я покажчика з попередньою зірочкою (чи групою зірочок). Формат оголошення покажчика :
Указатель - это адрес памяти, распределяемой для размещения идентификатора (в качестве идентификатора может выступать имя переменной, массива, структуры, строкового литерала). В том случае, если переменная объявлена как указатель, то она содержит адрес памяти, по которому может находится скалярная величина любого типа. При объявлении переменной типа указатель, необходимо определить тип объекта данных, адрес которых будет содержать переменная, и имя указателя с предшествующей звездочкой (или группой звездочек). Формат объявления указателя:

спецификатор-типа [модифікатор] * описувач .

Специфікатор-типа задає тип об'єкту і може бути будь-якого основного типу, типу структури, суміші (про це буде сказано нижче). Задаючи замість специфікатора-типу ключове слово void, можна своєрідним чином відстрочити специфікацію типу, на який посилається покажчик. Змінна, що оголошується як покажчик на тип void, може бути використана для посилання на об'єкт будь-якого типу. Проте для того, щоб можна було виконати арифметичні і логічні операції над покажчиками або над об'єктами, на які вони вказують, необхідно при виконанні кожної операції явно визначити тип об'єктів. Такі визначення типів може бути виконано за допомогою операції приведення типів. Спецификатор-типа задает тип объекта и может быть любого основного типа, типа структуры, смеси (об этом будет сказано ниже). Задавая вместо спецификатора-типа ключевое слово void, можно своеобразным образом отсрочить спецификацию типа, на который ссылается указатель. Переменная, объявляемая как указатель на тип void, может быть использована для ссылки на объект любого типа. Однако для того, чтобы можно было выполнить арифметические и логические операции над указателями или над объектами, на которые они указывают, необходимо при выполнении каждой операции явно определить тип объектов. Такие определения типов может быть выполнено с помощью операции приведения типов.

В якості модифікаторів при оголошенні покажчика можуть виступати ключові слова const, near, far, huge. Ключове слово const вказує, що покажчик не може бути змінений в програмі. Розмір змінної оголошеної як покажчик, залежить від архітектури комп'ютера і від використовуваної моделі пам'яті, для якої компілюватиметься програма. Покажчики на різні типи даних не обов'язково повинні мати однакову довжину.

Для модифікації розміру покажчика можна використати ключові слова near, far, huge.

Приклади:

```
unsigned int * a; /* змінна  a  є покажчиком
                  на тип unsigned int (цілі числа без знаку) */
double * x;       /* змінна  x  вказує  на  тип  даних з
                  плаваючою  точкою  подвоєної  точності      */
char * fuffer ;   /*  оголошується  покажчик з ім'ям fuffer
                   який вказує на  змінну типу char */
double nomer;
void *addres;
addres = & nomer;
(double *) addres ++;
 /* Змінна addres оголошена як покажчик на об'єкт будь-якого типу. Тому їй
можна присвоїти адресу будь-якого об'єкту (& - операція  обчислення адре-
си). Проте, як було відмічено вище, жодна арифмитическая операція не може
бути виконана над покажчиком,  поки
 не буде явно визначений тип даних,  на які він вказує. Це
 можна зробити,  використовуючи операцію приведення типу (double *) для
 перетворення addres до покажчика на тип double, а потім збільшення адреси.
*/
   const * dr;
 /* Змінна  dr  оголошена як покажчик на константне вираження, тобто зна-
чення покажчика може змінюватися в процесі виконання програми, а вели-
чина, на яку він вказує, ні. */
   unsigned char * const w = &obj.
 /* Змінна w оголошена як константний покажчик на дані типу char unsigned.
Це означає, що на протяг усієї програми
 w  вказуватиме на одну і ту ж область пам'яті.  Зміст же
 цій області може бути змінено. */
```

1.2.5. Змінні типу, що перераховує

Змінна, яка може набувати значення з деякого списку значень, називається змінною типу, що перераховує, або перерахуванням.

Оголошення перерахування розпочинається з ключового слова enum і має два формати представлення.

Формат 1. enum [имя-тега-перечисления] {список-перечисления} описувач[,описувач...];

Формат 2. enum имя-тега-перечисления описувач [,описувач.];

Оголошення перерахування задає тип змінної перерахування і визначає список іменованих констант, званий списком-перечисления. Значенням кожного імені списку є деяке ціле число.

Змінна типу перерахування може набувати значень однієї з іменованих констант списку. Іменовані константи списку мають тип int. Таким чином, пам'ять відповідає змінній перерахування, це пам'ять необхідна для розміщення значення типу int.

Змінна типу enum можуть використовуватися в індексних виразах і як операнди в арифметичних операціях і в операціях відношення.

У першому форматі 1 імена і значення перерахування задаються в списку перерахувань. Необов'язкове имя-тега-перечисления, це ідентифікатор, який іменує тег перерахування, визначений списком перерахування. Описувач іменує змінну перерахування. У оголошенні може бути задана більш ніж одна змінна типу перерахування.

Список-перечисления містить одну або декілька конструкцій виду :

ідентифікатор [= константне вираження]

Кожен ідентифікатор іменує елемент перерахування. Усі ідентифікатори в списку enum мають бути унікальними. У разі відсутності константного вираження першому ідентифікатору відповідає значення 0, наступному ідентифікатору - значення 1 і так далі. Ім'я константи перерахування еквівалентне її значенню.

Ідентифікатор, пов'язаний з константним вираженням, набуває значення, що задається цим константним виразом. Константне вираження повинне мати тип int і може бути як позитивним, так і негативним. Наступному ідентифікатору в списку привласнюється значення, рівне константному вираженню плюс 1, якщо цей ідентифікатор не має свого константного вираження. Використання елементів перерахування повинне підкорятися наступним правилам:

1. Змінна може містити значення, що повторюються.

2. Ідентифікатори в списку перерахування мають бути відмінні від усіх інших ідентифікаторів в тій же зоні видимості, включаючи імена звичайних змінних і ідентифікатори з інших списків перерахувань.

3. Імена типів перерахувань мають бути відмінні від інших імен типів перерахувань, структур і сумішей в цій же зоні видимості.

4. Значення може йти за останнім елементом списку перерахування.

Приклад:

```
enum week { SUB = 0,    /*    0   */
            VOS = 0,    /*    0   */
```

```
POND,      /*   1   */
VTOR,      /*   2   */
SRED,      /*   3   */
HETV,      /*   4   */
PJAT       /*   5   */
}   rab_ned ;
```

У цьому прикладі оголошений тег week, що перераховує, з відповідною безліччю значень, і оголошена змінна rab_ned що має тип week.

У другому форматі використовується ім'я тега перерахування для посилання на тип перерахування, визначуваний десь у іншому місці. Ім'я тега перерахування повинне відноситься до вже певного тега перерахування в межах поточної зони видимості. Оскільки тег перерахування оголошений десь у іншому місці, список перерахування не представлений в оголошенні.

Приклад:

enum week rab1;

У оголошенні покажчика на тип даних перерахування і оголошуваних typedef для типів перерахування можна використати ім'я тега перерахування до того, як цей тег перерахування визначений. Проте визначення перерахування повинне передувати будь-якій дії використовуваного покажчика на тип оголошення typedef. Оголошення без подальшого списку описувачів описує тег, або, якщо так можна сказати, шаблон перерахування. В объявлении указателя на тип данных перечисления и объявляемых typedef для типов перечисления можно использовать имя тега перечисления до того, как данный тег перечисления определен. Однако определение перечисления должно предшествовать любому действию используемого указателя на тип объявления typedef. Объявление без последующего списка описателей описывает тег, или, если так можно сказать, шаблон перечисления.

1.2.6. Масиви

Масиви - це група елементів однакового типу (double, float, int і тому подібне). З оголошення масиву компілятор повинен отримати інформацію про тип елементів масиву і їх кількість. Оголошення масиву має два формати:

спецификатор-типа описувач [константне - вираження];

спецификатор-типа описувач [ ];

Описувач - це ідентифікатор масиву .

Спецификатор-типа задає тип елементів оголошуваного масиву. Елементами масиву не можуть бути функції і елементи типу void.

Константное-выражение в квадратних дужках задає кількість елементів масиву. Константное-выражение при оголошенні масиву може бути опущено в наступних випадках:

- при оголошенні масив ініціалізувався,

- масив оголошений як формальний параметр функції,

- масив оголошений як посилання на масив, явно визначений в іншому файлі.

У мові СІ визначені тільки одновимірні масиви, але оскільки елементом масиву може бути масив, можна визначити і багатовимірні масиви. Вони формалізуються списком константных-выражений що йдуть за ідентифікатором масиву, причому кожне константное-выражение полягає у свої квадратні дужки.

Кожне константное-выражение в квадратних дужках визначає число елементів по цьому виміру масиву, так що оголошення двомірного масиву містить два константных-выражения, тривимірного, - три і так далі. Відмітимо, що в мові СІ перший елемент масиву має індекс рівний 0.

Приклади:

```
int a[2][3]; /* представлено у вигляді матриці
            a[0][0]  a[0][1] a[0][2]
            a[1][0]  a[1][1] a[1][2]      */
double b[10]; /* вектор з 10  елементів тих, що мають тип double */
int w[3][3] = { { 2, 3, 4 },
            { 3, 4, 8 },
            { 1, 0, 9 } };
```

У останньому прикладі оголошений масив w[3][3]. Списки, виділені у фігурні дужки, відповідають рядкам масиву, у разі відсутності дужок ініціалізація буде виконана неправильно.

У мові СІ можна використати перерізи масиву, як і в інших мовах високого рівня (PL1 і тому подібне), проте на використання перерізів накладається ряд обмежень. Перерізи формуються внаслідок опускання однієї або декількох пар квадратних дужок. Пари квадратних дужок можна відкидати тільки справа наліво і строго послідовно. Перерізи масивів використовуються при організації обчислювального процесу у функціях мови СІ, що розробляються користувачем.

Приклади:

```
int s[2][3];
```

Якщо при зверненні до деякої функції написати s[0], те передаватиметься нульовий рядок масиву s.

int b[2][3][4];

При зверненні до масиву b можна написати, наприклад, b[1][2] і передаватиметься вектор з чотирьох елементів, а звернення b[1] дасть двомірний масив розміром 3 на 4. Не можна написати b[2][4], маючи на увазі, що передаватися буде вектор, тому що це не відповідає обмеженню накладеному на використання перерізів масиву.

Приклад оголошення символьного масиву.

char str[] = "оголошення символьного масиву";

Слід враховувати, що в символьному літералі знаходиться на один елемент більше, оскільки останній з елементів є послідовністю '\, що управляє, 0'.

1.2.7. Структури

Структуры - це складений об'єкт, в який входять елементи будь-яких типів, за винятком функцій. На відміну від масиву, який є однорідним об'єктом, структура може бути неоднорідною. Тип структури визначається записом виду :

struct { список визначень }

У структурі обов'язково має бути вказаний хоч би один компонент. Визначення структур має наступний вигляд:

тип-данных описувач;

де тип-данных вказує тип структури для об'єктів, визначуваних в описувачах. У простій формі описувачі є ідентифікаторами або масивами.

Приклад:

```
struct { double x, y; } s1, s2, sm[9];
struct {  int   year;
          char  moth, day; } date1, date2;
```

Змінні s1, s2 визначаються як структури, кожна з яких складається з двох компонент x і y. Змінна sm визначається як масив з дев'яти структур. Кожна з двох змінних date1, date2 складається з трьох компонентів year, moth, day. >p>Існує і інший спосіб асоціювання імені з типом структури, він заснований

на використанні тега структури. Тег структури аналогічний тегу типу, що перераховує. Тег структури визначається таким чином:

struct тег { список описів; };

де тег є ідентифікатором.

У наведеному нижче прикладі ідентифікатор student описується як тег структури :

```
struct student { char name[25];
                 int  id, age;
                 char  prp;        };
```

Тег структури використовується для подальшого оголошення структур цього виду у формі:

struct тег список-идентификаторов;

Приклад:

struct studeut st1, st2;

Використання тегів структури потрібне для опису рекурсивних структур. Нижче розглядається використання рекурсивних тегів структури.

```
struct node { int data;
              struct node * next; }   st1_node;
```

Тег структури node дійсно є рекурсивним, оскільки він використовується у своєму власному описі, тобто у формалізації покажчика next. Структури не можуть бути прямо рекурсивними, тобто структура node не може містити компоненту, що є структурою node, але будь-яка структура може мати компоненту, що є покажчиком на свій тип, як і зроблено в наведеному прикладі.

Доступ до компонент структури здійснюється за допомогою вказівки імені структури і наступного через точку імені виділеного компонента, наприклад:

```
st1.name="Іванов";
st2.id=st1.id;
st1_node.data=st1.age;
```

1.2.8. Об'єднання (суміші)

Об'єднання подібно до структури, проте в кожен момент часу може використовуватися (чи іншими словами бути відповіддю) тільки один з елементів об'єднання. Тип об'єднання може задаватися в наступному виді:

```
union {   опис елементу 1;
          ...
          опис елементу n; };
```

Головною особливістю об'єднання є те, що для кожного з оголошених елементів виділяється одна і та ж область пам'яті, тобто вони перекриваються. Хоча доступ до цієї області пам'яті можливий з використанням будь-якого з елементів, елемент для цієї мети повинен вибиратися так, щоб отриманий результат не був безглуздим.

Доступ до елементів об'єднання здійснюється тим же способом, що і до структур. Тег об'єднання може бути формалізований точно так, як і тег структури.

Об'єднання застосовується для наступних цілей:

- ініціалізації використовуваного об'єкту пам'яті, якщо в кожен момент часу тільки один об'єкт з багатьох є активним;

- інтерпретації основного представлення об'єкту одного типу, начебто цьому об'єкту був присвоєний інший тип.

Пам'ять, яка відповідає змінній типу об'єднання, визначається величиною, необхідною для розміщення найбільш довгого елементу об'єднання. Коли використовується елемент меншої довжини, то змінна типу об'єднання може містити невживану пам'ять. Усі елементи об'єднання зберігаються в одній і тій же області пам'яті, починаючи з однієї адреси.

Приклад:

```
union {   char  fio[30];
          char  adres[80];
          int   vozrast;
          int   telefon;   } inform;
union {   int ax;
          char al[2];      }   ua;
```

При використанні об'єкту infor типу union можна обробляти тільки той елемент який отримав значення, тобто після привласнення значення елементу inform.fio, не має сенсу звертатися до інших елементів. Об'єднання ua дозво-

ляє отримати окремий доступ до молодшому ua.al[0] і до старшому ua.al[1] байтам двобайтового числа ua.ax .

1.2.9. Поля бітів

Елементом структури може бути бітове поле, що забезпечує доступ до окремих біт пам'яті. Поза структурами бітові поля оголошувати не можна. Не можна також організовувати масиви бітових полів і не можна застосовувати до полів операцію визначення адреси. У загальному випадку тип структури з бітовим полем задається в наступному виді:

```
struct { unsigned ідентифікатор 1:  длина-поля  1;
         unsigned ідентифікатор 2:  длина-поля  2;    }
```

довжина - поля задається цілим вираженням або константою. Ця константа визначає число бітів, відведене відповідному полю. Полі нульової довжини означає вирівнювання на границю наступного слова.

Приклад:

```
struct { unsigned a1:  1;
         unsigned a2:  2;
         unsigned a3:  5;
         unsigned a4:  2;  } prim;
```

Структури бітових полів можуть містити і знакові компоненти. Такі компоненти автоматично розміщуються на відповідних межах слів, при цьому деякі біти слів можуть залишатися невикористаними.

Посилання на поле бітів виконуються точно так, як і компоненти загальних структур. Саме ж бітове поле розглядається як ціле число, максимальне значення якого визначається завдовжки поля.

1.2.10. Змінні зі змінюваною структурою

Дуже часто деякі об'єкти програми відносяться до одного і тому ж класу, відрізняючись лише деякими деталями. Розглянемо, наприклад, представлення геометричних фігур. Загальна інформація про фігури може включати такі елементи, як площу, периметр. Проте відповідна інформація про геометричні розміри може виявитися різною залежно від їх форми. Очень часто некоторые объекты программы относятся к одному и тому же классу, отличаясь лишь некоторыми деталями. Рассмотрим, например, представление геометрических фигур. Общая информация о фигурах может включать такие элементы, как площадь, периметр. Однако соответствующая информация о геометрических размерах может оказаться различной в зависимости от их формы.

Розглянемо приклад, в якому інформація про геометричні фігури представляється на основі комбінованого використання структури і об'єднання.

```
struct figure {
        double area, perimetr;  /* загальних компонент       */
        int type;               /* ознака компонента     */
        union                   /* перерахування компонент */
            { double radius;  /* коло            */
              double a[2];    /* прямокутник         */
              double b[3];    /* трикутник           */
            } geom_fig;
            }  fig1, fig2  ;
```

У загальному випадку кожен об'єкт типу figure складатиметься з трьох компонентів: area, perimetr, type. Компонент type називається міткою активного компонента, оскільки він використовується для вказівки, який з компонентів об'єднання geom_fig є активним в даний момент. Така структура називається змінною структурою, тому що її компоненти міняються залежно від значення мітки активного компонента (значення type).

Відмітимо, що замість компоненти type типу int, доцільно було б використати перераховуваний тип. Наприклад, такий

```
enum figure_chess { CIRCLE
BOX
TRIANGLE    } ;
```

Константи CIRCLE, BOX, TRIANGLE отримають значення відповідно рівні 0, 1, 2. Змінна type може бути оголошена як що має тип, що перераховує :

```
enum figure_chess type;
```

В цьому випадку компілятор CI попередить програміста про потенційно помилкові привласнення, таких, наприклад, як

```
figure.type = 40;
```

У загальному випадку змінна структури складатиметься з трьох частин: набір загальних компонент, мітки активного компонента і частини з компонентами, що міняються. Загальна форма змінної структури, має наступний вигляд:

```
struct { загальні компоненти;
        мітка активного компонента;
          union { опис компоненти 1 ;
                  опис компоненти 2 ;
                  :::
                   опис компоненти n ;
                  } идентификатор-объединения ;
        } идентификатор-структуры ;
```

Приклад визначення змінної структури з ім'ям helth_record

```
struct { /* загальна інформація */
        char  name [25];   /* ім'я       */
        int   age;         /* вік   */
        char  sex;         /* поли       */
        /*   мітка   активного   компонента  */
        /*   (сімейний стан)           */
        enum merital_status ins;
        /* змінна частина */
        union  {   /* неодружений        */
                /* немає компонент */
               struct {  /* перебуває в шлюбі */
                      char marripge_date[8];
                      char spouse_name[25];
                      int  no_children;
                     }  marriage_info;
                /* розлучений */
                char date_divorced[8];
               }   marital_info;
         } health_record;

  enum marital_status { SINGLE, /* неодружений   */
                MARRIGO, /* одружений    */
                DIVOREED /* розлучений */
                } ;
```

Звертатися до компонент структури можна за допомогою посилань:

```
helth_record.neme
helth_record.ins
helth_record.marriage_info.marriage_date .
```

1.2.11. Визначення об'єктів і типів

Як вже говорилося вище, усі змінні використовувані в програмах на мові СІ, мають бути оголошені. Тип оголошуваної змінної залежить від того, яке ключове слово використовується як специфікатор типу і чи являється описувач простим ідентифікатором або ж комбінацією ідентифікатора з модифікатором покажчика (зірочка), масиву (квадратні дужки) або функції (круглі дужки).

При оголошенні простої змінної, структури, суміші або об'єднання, а також перерахування, описувач - це простий ідентифікатор. Для оголошення покажчика, масиву або функції ідентифікатор модифікується відповідним чином: зірочкою ліворуч, квадратними або круглими дужками справа.

Відмітимо важливу особливість мови СІ, при оголошенні можна використати одночасно більше за один модифікатор, що дає можливість створювати безліч різних складних описувачів типів.

Проте потрібно пам'ятати, що деякі комбінації модифікаторів недопустимі:

- елементами масивів не можуть бути функції,

- функції не можуть повертати масиви або функції.

При ініціалізації складних описувачів квадратні і круглі дужки (праворуч від ідентифікатора) мають пріоритет перед зірочкою (зліва від ідентифікатора). Квадратні або круглі дужки мають один і той же пріоритет і розкриваються зліва направо. Специфікатор типу розглядається на останньому кроці, коли описувач вже повністю проінтерпретований. Можна використати круглі дужки, щоб поміняти порядок інтерпретації на необхідний.
При инициализации сложных описателей квадратные и круглые скобки (справа от идентификатора) имеют приоритет перед звездочкой (слева от идентификатора). Квадратные или круглые скобки имеют один и тот же приоритет и раскрываются слева направо. Спецификатор типа рассматривается на последнем шаге, когда описатель уже полностью проинтерпретирован. Можно использовать круглые скобки, чтобы поменять порядок интерпретации на необходимый.

Для інтерпретації складних описів пропонується просте правило, яке звучить як "зсередини назовні", і складається з чотирьох кроків.

1. Розпочати з ідентифікатора і подивитися управо, чи є квадратні або круглі дужки.

2. Якщо вони є, то проінтерпретувати цю частину описувача і потім подивитися наліво в пошуку зірочки.

3. Якщо на будь-якій стадії справа зустрінеться закриваюча кругла дужка, то спочатку необхідно застосувати усі ці правила усередині круглих дужок, а потім продовжити інтерпретацію.

4. Інтерпретувати специфікатор типу.

Приклади:

```
int    * ( * comp [10]) ();
 6     5   3   1    2    4
```

У цьому прикладі оголошується змінна comp (1), як масив з десяти (2) покажчиків (3) на функції (4), що повертають покажчики (5) на цілі значення (6).

```
char  * ( * ( * var ) () )[10];
  7   6   4   2  1     3     5
```

Змінна var (1) оголошена як покажчик (2) на функцію (3) що повертає покажчик (4) на масив (5) з 10 елементів, які є покажчиками (6) на значення типу char.

Окрім оголошень змінних різних типів, є можливість оголосити типи. Це можна зробити двома способами. Перший спосіб - вказати ім'я тега при оголошенні структури, об'єднання або перерахування, а потім використати це ім'я в оголошенні змінних і функцій в якості посилання на цей тег. Другий - використати для оголошення типу ключове слово typedef.
Кроме объявлений переменных различных типов, имеется возможность объявить типы. Это можно сделать двумя способами. Первый способ - указать имя тега при объявлении структуры, объединения или перечисления, а затем использовать это имя в объявлении переменных и функций в качестве ссылки на этот тег. Второй - использовать для объявления типа ключевое слово typedef.

При оголошенні з ключовим словом typedef, ідентифікатор стоїть на місці описуваного об'єкту, є ім'ям типу даних, що вводиться в розгляд, і далі цей тип може бути використаний для оголошення змінних.

Відмітимо, що будь-який тип може бути оголошений з використанням ключового слова typedef, включаючи типи покажчика, функції або масиву. Ім'я з ключовим словом typedef для типів покажчика, структури, об'єднання може бути оголошене перш ніж ці типи будуть визначені, але в межах видимості об'явника.

Приклади:

```
typedef double (* MATH) ( );
  /* MATH - нове ім'я типу, що представляє покажчик на
  функцію, що повертає значення типу double          */
  MATH cos;
  /* cos  покажчик на функцію, що повертає
  значення типу  double                              */
  /* Можна провести еквівалентне оголошення          */
  double (* cos) ( );

typedef char FIO[40]
   /* FIO - масив з сорока символів                  */
FIO person;
   /*  Змінна person  -  масив з сорока символів  */
   /*  Це  еквівалентно оголошенню                   */
char person[40];
```

При оголошенні змінних і типів тут були використані імена типів (MATH FIO). Окрім цього, імена типів можуть ще використовуватися в трьох випадках: в списку формальних параметрів, в оголошенні функцій, в операціях приведення типів і в операції sizeof (операція приведення типу).

Іменами типів для основних типів, типів перерахування, структури і суміші являються специфікатори типів для цих типів. Імена типів для типів покажчика масиву і функції задаються за допомогою абстрактних описувачів таким чином:

специфікатор-типа абстрактный-описатель;

Абстрактный-описатель - це описувач без ідентифікатора, що складається з одного або більше за модифікатори покажчика, масиву або функції. Модифікатор покажчика (*) завжди задається перед ідентифікатором в описувачі, а модифікатори масиву [] і функції () - після нього. Так, щоб правильно інтерпретувати абстрактний описувач, треба розпочати інтерпретацію з ідентифікатора, що мається на увазі.

Абстрактні описувачі можуть бути складними. Дужки в складних абстрактних описувачі задають порядок інтерпретації подібно до того, як це робилося при інтерпретації складних описувачів в оголошеннях.

1.2.12. Ініціалізація даних

При оголошенні змінної їй можна присвоїти початкове значення, приєднуючи ініціатор до описувача. Ініціатор розпочинається зі знаку "=" і має наступні форми.

Формат 1: = ініціатор;

Формат 2: ={ список - ініціаторів };

Формат 1 використовується при ініціалізації змінних основних типів і покажчиків, а формат 2 - при ініціалізації складених об'єктів.

Приклади:

char tol = 'N';

Змінна tol ініціалізувалася символом 'N'.

const long megabute = (1024 * 1024);

Змінна megabute, що Не модифікується, ініціалізувалася константним вираженням після чого вона не може бути змінена.

static int b[2][2] = {1,2,3,4};

Ініціалізувався двомірний масив b цілих величин елементам масиву привласнюються значення зі списку. Ця ж ініціалізація може бути виконана таким чином:

static int b[2][2] = { { 1,2 }, { 3,4 } };

При ініціалізації масиву можна опустити одну або декілька розмірності

static int b[3[] = { { 1,2 }, { 3,4 } };

Якщо при ініціалізації вказані менше значень для рядків, то елементи, що залишилися, ініціалізувалися 0, тобто при описі

static int b[2][2] = { { 1,2 }, { 3 } };

елементи першого рядка отримають значення 1 і 2, а другий 3 і 0.

При ініціалізації складених об'єктів, треба уважно стежити за використанням дужок і списків ініціалізаторів.

Приклади:

```
struct complex { double real;
                 double imag; }  comp [2][3] =
    { { {1,1}, {2,3}, {4,5} },
      { {6,7}, {8,9}, {10,11} } };
```

У цьому прикладі ініціалізувався масив структур comp з двох рядків і трьох стовпців, де кожна структура складається з двох елементів real і imag.

```
struct complex comp2 [2][3] =
  { {1,1},{2,3},{4,5}, {6,7},{8,9},{10,11} };
```

В даному прикладі компілятор інтерпретує дані фігурні дужки таким чином:

- перша ліва фігурна дужка - початок складеного ініціатора для масиву comp2;

- друга ліва фігурна дужка - початок ініціалізації першого рядка масиву comp2[0]. Значення 1,1 привласнюються двом елементам першої структури;

- перша права дужка (після 1) вказує компілятору, що список ініціаторів для рядка масиву закінчений, і елементи структур, що залишилися, в рядку comp[0] автоматично ініціалізувалися нулем;

- аналогічно список {2,3} ініціалізував першу структуру в рядку comp[1], а структури масиву, що залишилися, перетворюються на нулі;

- на наступний список ініціалізаторів {4,5} компілятор повідомлятиме про можливу помилку оскільки рядок 3 в масиві comp2 відсутній.

При ініціалізації об'єднання задається значення першого елементу об'єднання відповідно до його типу.

Приклад:

```
union tab { unsigned char name[10];
          int tab1;
          }          pers  = {'A ',' H ',' T ',' O ',' H'};
```

Ініціалізувалася змінна pers.name, і оскільки це масив, для його ініціалізації потрібно список значень у фігурних дужках. Перші п'ять елементів масиву ініціалізувалися значеннями зі списку, інші нулями.

Ініціалізацію масиву символів можна виконати шляхом використання строкового літерала.

```
char stroka[ ] = "привіт";
```

Ініціалізувався масив символів з 7 елементів, останнім елементом (сьомим) буде символ '\0 ', яким завершуються усі строкові літерали.

У тому випадку, якщо задається розмір масиву, а строковий літерал довший, ніж розмір масиву, то зайві символи відкидаються.

Наступне оголошення ініціалізувало змінну stroka як масив, що складається з семи елементів.

```
char stroka[5] = "привіт";
```

У змінну stroka потрапляють перші п'ять елементів літерала, а символи 'Т' і '\0' відкидаються.

Якщо рядок коротший, ніж розмір масиву, то елементи масиву, що залишилися, заповнюються нулями.

Відмітимо, що ініціалізація змінної типу tab може мати наступний вигляд:

```
union tab pers1 = "Антон";
```

і, таким чином, в символьний масив потраплять символи:

'A','H','T','O','H','\0

а інші елементи ініціалізують нулем.

1.3. Вирази І Привласнення

1.3.1. Операнди і операції

Комбінація знаків операцій і операндів, результатом якої є певне значення, називається вираженням. Знаки операцій визначають дії, які мають бути виконані над операндами. Кожен операнд у вираженні може бути вираженням. Значення вираження залежить від розташування знаків операцій і круглих дужок у вираженні, а також від пріоритету виконання операцій. Комбинация знаков операций и операндов, результатом которой является определенное значение, называется выражением. Знаки операций определяют действия, которые должны быть выполнены над операндами. Каждый операнд в выражении может быть выражением. Значение выражения зависит от расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций.

У мові СІ привласнення також є вираженням, і значенням такого вираження є величина, яка привласнюється.

При обчисленні виразів тип кожного операнда може бути перетворений до іншого типу. Перетворення типів можуть бути неявними, при виконанні операцій і викликів функцій, або явними, при виконанні операцій приведення типів.

Операнд - це константа, літерал, ідентифікатор, виклик функції, індексне вираження, вираження вибору елементу або складніше вираження, сформоване комбінацією операндів, знаків операцій і круглих дужок. Будь-який операнд, який має константне значення, називається константним вираженням. Кожен операнд має тип.

Якщо як операнд використовується константа, то йому відповідає значення і тип константи, що представляє його. Ціла константа може бути типу int, long, unsigned int, unsigned long, залежно від її значення і від форми запису. Символьна константа має тип int. Константа з плаваючою точкою завжди має тип double.

Строковий літерал складається з послідовності символів, поміщених в лапки, і представляється в пам'яті як масив елементів типу char, що ініціалізувався вказаною послідовністю символів. Значенням строкового літерала є адреса першого елементу рядка і синтаксично строковий літерал є покажчиком, що не модифікується, на тип char. Строкові літерали можуть бути використані в якості операндів у виразах, що допускають величини типу покажчиків. Проте оскільки рядки не є змінними, їх не можна використати в лівій частині операції привласнення. Строковый литерал состоит из последовательности символов, заключенных в кавычки, и представляется в памяти как массив элементов типа char, инициализируемый указанной последовательностью символов. Значением строкового литерала является адрес первого элемента строки и синтаксически строковый литерал является немодифицируемым указателем на тип char. Строковые литералы могут быть использованы в качестве операндов в выражениях, допускающих величины типа указателей. Однако так как

строки не являются переменными, их нельзя использовать в левой части операции присваивания.

Слід пам'ятати, що останнім символом рядка завжди є нульовий символ, який автоматично додається при зберіганні рядка в пам'яті.

Ідентифікатори змінних і функцій. Кожен ідентифікатор має тип, який встановлюється при його оголошенні. Значення ідентифікатора залежить від типу таким чином:

- ідентифікатори об'єктів цілих і плаваючих типів представляють значення відповідного типу;

- ідентифікатор об'єкту типу enum представлений значенням однієї константи з безлічі значень констант в перерахуванні. Значенням ідентифікатора є константне значення. Тип значення є int, що виходить з визначення перерахування;

- ідентифікатор об'єкту типу struct або union представляє значення, визначене структурою або об'єднанням;

- ідентифікатор, що оголошується як покажчик, представляє покажчик на значення, задане в оголошенні типу;

- ідентифікатор, що оголошується як масив, представляє покажчик, значення якого є адресою першого елементу масиву. Тип величин, що адресуються покажчиком, - це тип елементів масиву. Відмітимо, що адреса масиву не може бути змінена під час виконання програми, хоча значення окремих елементів може змінюватися. Значення покажчика, що представляється ідентифікатором масиву, не є змінною і тому ідентифікатор масиву не може з'являтися в лівій частині оператора привласнення. идентификатор, объявляемый как массив, представляет указатель, значение которого является адресом первого элемента массива. Тип адресуемых указателем величин - это тип элементов массива. Отметим, что адрес массива не может быть изменен во время выполнения программы, хотя значение отдельных элементов может изменяться. Значение указателя, представляемое идентификатором массива, не является переменной и поэтому идентификатор массива не может появляться в левой части оператора присваивания.

- ідентифікатор, що оголошується як функція, представляє покажчик, значення якого є адресою функції, що повертає значення певного типу. Адреса функції не змінюється під час виконання програми, міняється тільки повертане значення. Таким чином, ідентифікатори функцій не можуть з'являтися в лівій частині операції привласнення.

Виклик функцій складається з вираження, за яким йде необов'язковий список виразів в круглих дужках :

вираження-1 ([список виразів])

Значенням вираження-1 має бути адреса функції (наприклад, ідентифікатор функції). Значення кожного вираження зі списку виразів передається у функцію в якості фактичного аргументу. Операнд, що є викликом функції, має тип і значення повертаного функцією значення.

Індексне вираження задає елемент масиву і має вигляд:

вираження-1 [вираження-2]

Тип індексного вираження є типом елементів масиву, а значення представляє величину, адреса якої обчислюється за допомогою значень вираження-1 і вираження-2.

Звичайне вираження-1 - це покажчик, наприклад, ідентифікатор масиву, а вираження-2 - це ціла величина. Проте вимагається тільки, щоб один з виразів був покажчиком, а друге цілочисельною величиною. Тому вираження-1 може бути цілочисельною величиною, а вираження-2 покажчиком. У будь-якому випадку вираження-2 повинно бути поміщено в квадратні дужки. Хоча індексне вираження зазвичай використовується для посилань на елементи масиву, проте індекс може з'являтися з будь-яким покажчиком. Обычно выражение-1 - это указатель, например, идентификатор массива, а выражение-2 - это целая величина. Однако требуется только, чтобы одно из выражений было указателем, а второе целочисленной величиной. Поэтому выражение-1 может быть целочисленной величиной, а выражение-2 указателем. В любом случае выражение-2 должно быть заключено в квадратные скобки. Хотя индексное выражение обычно используется для ссылок на элементы массива, тем не менее индекс может появляться с любым указателем.

Індексні вирази для посилання на елементи одновимірного масиву обчислюються шляхом складання цілої величини зі значеннями покажчика з подальшим застосуванням до результату операції разадресации (*).

Оскільки один з виразів, вказаних в індексному вираженні, є покажчиком, то при складанні використовуються правила адресної арифметики, згідно з якими ціла величина перетвориться до адресного представлення, шляхом множення її на розмір типу, що адресується покажчиком. Нехай, наприклад, ідентифікатор arr оголошений як масив елементів типу double.

double arr[10];

Так, щоб отримати доступ до i -тому елементу масиву arr можна написати arr[i], що, в силу сказаного вище, еквівалентно i[a]. При цьому величина i множиться на розмір типу double і є адресою i -го елементу масиву arr від його початку. Потім це значення складається зі значенням покажчика arr, що у свою чергу дає адресу i -го елементу масиву. До отриманої адреси застосовується операція разадресации, тобто здійснюється вибірка елементу масиву arr за сформованою адресою. Таким образом, чтобы получить доступ к i-тому элементу массива arr можно написать arr[i], что, в силу сказанного выше, эквивалентно i[a]. При этом величина i умножается на размер типа double и представляет собой адрес i-го элемента массива arr от его начала.

Затем это значение складывается со значением указателя arr, что в свою очередь дает адрес i-го элемента массива. К полученному адресу применяется операция разадресации, т.е. осуществляется выборка элемента массива arr по сформированному адресу.

Таким чином, результатом індексного вираження arr[i] (чи i[arr]) являється значення і -го елементу масиву.

Вираження з декількома індексами посилається на елементи багатовимірних масивів. Багатовимірний масив - це масив, елементами якого є масиви. Наприклад, першим елементом тривимірного масиву є масив з двома вимірами.

Для посилання на елемент багатовимірного масиву індексне вираження повинне мати декілька індексів ув'язнених квадратні дужки:

вираження-1 [вираження-2][вираження-3] ...

Таке індексне вираження інтерпретується зліва направо, тобто спочатку розглядається перше індексне вираження:

вираження-1 [вираження-2]

Результат цього виразу є адресне вираження, з яким складається вираження-3 і так далі. Операція разадресации здійснюється після обчислення останнього індексного вираження. Відмітимо, що операція разадресации не застосовується, якщо значення останнього покажчика адресує величину типу масиву.

Приклад:

int mass [2][5][3];

Розглянемо процес обчислення індексного вираження mass[1][2][2].

1. Обчислюється вирази mass[1]. Посилання індекс 1 множиться на розмір елементу цього масиву, елементом же цього масиву є двомірний масив що містить 5x3 елементів, що мають тип int. Значення, що набуває, складається зі значенням покажчика mass. Результат являється покажчик на другий двомірний масив розміром (5x3) в тривимірному масиві mass.

2. Другий індекс 2 вказує на розмір масиву з трьох елементів типу int, і складається з адресою, відповідним mass [1].

3. Оскільки кожен елемент тривимірного масиву - це величина типу int, то індекс 2 збільшується на розмір типу int перед складанням з адресою mass [1][2].

4. Нарешті, виконується разадресация отриманого покажчика. Результуючим вираженням буде елемент типу int.

Якщо було б вказано mass [1][2], те результатом був би покажчик на масив з трьох елементів типу int. Відповідно значенням індексного вираження mass [1] являється покажчик на двомірний масив.

Вираження вибору елементу застосовується, якщо в якості операнда потрібно використати елемент структури або об'єднання. Таке вираження має значення і тип вибраного елементу. Розглянемо дві форми вираження вибору елементу :

вираження.ідентифікатор,

вираження->ідентифікатор .

У першій формі вираження представляє величину типу struct або union, а ідентифікатор - це ім'я елементу структури або об'єднання. У другій формі вираження повинне мати значення адреси структури або об'єднання, а ідентифікатор - ім'ям вибираного елементу структури або об'єднання.

Обидві форми вираження вибору елементу дають однаковий результат. Дійсно, запис, що включає знак операції вибору (->), є скороченою версією запису з точкою для випадку, коли вираженню що стоїть перед точкою передує операція разадресации (*), застосована до покажчика, тобто запис

вираження -> ідентифікатор

еквівалентна запису

(* вираження). ідентифікатор

у разі, якщо вираження є покажчиком.

Приклад:

```
struct tree { float        num;
              int          spisoc[5];
              struct tree  *left;     }  tr[5], elem ;
      elem.left = & elem;
```

У наведеному прикладі використовується операція вибору (.) для доступу до елементу left структурної змінної elem. Таким чином елементу left структурної змінної elem привласнюється адреса самої змінної elem, тобто змінна elem зберігає посилання на себе саму.

Приведення типів ця зміна (перетворення) типу об'єкту. Для виконання перетворення необхідно перед об'єктом записати в дужках потрібний тип:

(имя-типа) операнд.

Приведення типів використовуються для перетворення об'єктів одного скалярного типу в інший скалярний тип. Проте вираженню з приведенням типу не може бути присвоєне інше значення.

Приклад:

```
int i;
bouble x;
b = (double) i+2.0;
```

В даному прикладі ціла змінна і за допомогою операції приведення типів наводиться до плаваючого типу, а потім вже бере участь в обчисленні вираження.

Константне вираження - цей вираз, результатом якого є константа. Операндом константного вираження можуть бути цілі константи, символьні константи, константи з плаваючою точкою, константи перерахування, вираження приведення типів, вираження з операцією sizeof і інші константні вирази. Проте на використання знаків операцій в константних виразах накладаються наступні обмеження: Константное выражение - это выражение, результатом которого является константа. Операндом константного выражения могут быть целые константы, символьные константы, константы с плавающей точкой, константы перечисления, выражения приведения типов, выражения с операцией sizeof и другие константные выражения. Однако на использование знаков операций в константных выражениях налагаются следующие ограничения:

1. У константних виразах не можна використати операції привласнення і послідовного обчислення (,).

2. Операція "адреса" (&) може бути використана тільки при деяких ініціалізаціях.

Вираження зі знаками операцій можуть брати участь у виразах як операнди. Вираження зі знаками операцій можуть бути унарними (з одним операндом), бінарними (з двома операндами) і тернарными (з трьома операндами).

Унарне вираження складається з операнда і передування йому знаку унарної операції і має наступний формат:

знак-унарной-операции операнд .

Бінарне вирази складається з двох операндів, розділених знаком бінарної операції :

операнд1 знак-бинарной-операции операнд2 .

Тернарное вираження складається з трьох операндів, розділених знаками тернарной операції (?) і (:), і має формат:

операнд1 ? операнд2: операнд3 .

Операції. По кількості операндів, що беруть участь в операції, операції підрозділяються на унарні, бінарні і тернарные.

У мові Сі є наступні унарні операції:

- арифметичне заперечення (заперечення і доповнення);

~ побітове логічне заперечення (доповнення);

! логічне заперечення;

* разадресация (непряма адресація);

& обчислення адреси;

+ унарний плюс;

++ збільшення (інкремент);

-- зменшення (декремент);

sizeof розмір .

Унарні операції виконуються справа наліво.

Операції збільшення і зменшення збільшують або зменшують значення операнда на одиницю і можуть бути записані як справа так і зліва від операнда. Якщо знак операції записаний перед операндом (префіксна форма), то зміна операнда відбувається до його використання у вираженні. Якщо знак операції записаний після операнда (постфіксна форма), то операнд спочатку використовується у вираженні, а потім відбувається його зміна.

На відміну від унарних, бінарні операції, список яких приведений в таблицю.7, виконуються зліва направо.

Таблиця 7

Знак	**Операція**	**Група операцій**

операції		
*	Множення	Мультиплікативні
/	Ділення	
%	Залишок від ділення	
+	Складання	Адитивні
-	Віднімання	
<<	Зрушення вліво	Операції зрушення
>>	Зрушення управо	
<	Менше	Операції відношення
<=	Менше або рівно	
>=	Більше або рівно	
==	Рівно	
!=	Не рівно	
&	Порозрядне І	Порозрядні операції
|	Порозрядне АБО	
^	Що порозрядне, що виключає АБО	
&&	Логічне І	Логічні операції
||	Логічне АБО	
’	Послідовне обчислення	Послідовного обчислення
=	Привласнення	Операції привласнення
*=	Множення з привласненням	
/=	Ділення з привласненням	
%=	Залишок від ділення з привласненням	
-=	Віднімання з привласненням	
+=	Складання з привласненням	
<<=	Зрушення вліво з привласненням	

>>=	Зрушення управо привласненням	
&=	Порозрядне І з привласненням	
|=	Порозрядне АБО з привласненням	
^=	Що порозрядне, що виключає АБО з привласненням	

Лівий операнд операції привласнення має бути вираженням, що посилається на область пам'яті (але не об'єктом оголошеним з ключовим словом const), такі вирази називаються леводопустимыми до них відносяться:

- ідентифікатори даних цілого і плаваючого типів, типів покажчика, структури, об'єднання;

- індексні вирази, виключаючи вирази що мають тип масиву або функції;

- вирази вибору елементу (->) і (.), якщо вибраний елемент є леводопустимым;

- вирази унарної операції разадресации (*), за винятком виразів, що посилаються на масив або функцію;

- вираження приведення типу якщо результуючий тип не перевищує розміру первинного типу.

При записі виразів слід пам'ятати, що символи (*), (&), (!), (+) можуть\ означати унарну або бінарну операцію.

1.3.2. Перетворення при обчисленні виразів

При виконанні операцій робиться автоматичне перетворення типів, щоб привести операнди виразів до загального типу або щоб розширити короткі величини до розміру цілих величин, використовуваних в машинних командах. Виконання перетворення залежить від специфіки операцій і від типу операнда або операндів.

Розглянемо загальні арифметичні перетворення.

1. Операнди типу float перетворяться до типу double.

2. Якщо один операнд long double, то другою перетвориться до цього ж типу.

3. Якщо один операнд double, то другий також перетвориться до типу double.

4. Будь-які операнди типу char і short перетворяться до типу int.

5. Будь-які операнди unsigned char або unsigned short перетворяться до типу unsigned int.

6. Якщо один операнд типу unsigned long, то другою перетвориться до типу unsigned long.

7. Якщо один операнд типу long, то другою перетвориться до типу long.

8. Якщо один операнд типу unsigned int, то другий операнд перетвориться до цього ж типу.

Таким чином, можна відмітити, що при обчисленні виразів операнди перетворяться до типу того операнда, який має найбільший розмір.

Приклад:

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

При виконанні оператора привласнення правила перетворення використовуватимуться таким чином. Операнд ch перетвориться до unsigned int (правило 5). Потім він перетвориться до типу unsigned long (правило 6). За цим же правилом і перетвориться до unsigned long і результат операції, поміщеної в круглі дужки матиме тип unsigned long. Потім він перетвориться до типу double (правило 3) і результат усього вираження матиме тип double.

1.3.3. Операції заперечення і доповнення

Операція арифметичного заперечення (-) виробляє заперечення свого операнда. Операнд має бути цілою або плаваючою величиною. При виконанні здійснюються звичайні арифметичні перетворення.

Приклад:

```
double  u = 5;
u = - u;          /* змінній u привласнюється її заперечення
                     тобто u  набуває значення - 5     */
```

Операція логічного заперечення "НЕ" (!) виробляє значення 0, якщо операнд є істина (не нуль), і значення 1, якщо операнд дорівнює нулю (0). Результат має тип int. Операнд має бути цілого або плаваючого типу або типу покажчик.

Приклад:

```
int t, z=0;
t=!z;
```

Змінна t отримає значення рівне 1, оскільки змінна z мала значення рівне 0 (неправдиво).

Операція двійкового доповнення (~) виробляє двійкове доповнення свого операнда. Операнд має бути цілого типу. Здійснюється звичайне арифметичне перетворення, результат має тип операнда після перетворення.

Приклад:

```
char          b = '9';
unsigned char   f;
 b = ~f;
```

Шістнадцятиричне значення символу '9' рівне 39. В результаті операції ~f буде отримане шістнадцятиричне значення С6, що відповідає символу 'ц'.

1.3.4. Операції разадресации і адреси

Ці операції використовуються для роботи зі змінними типу покажчик.

Операція разадресации (*) здійснює непрямий доступ до величини, що адресується, через покажчик. Операнд має бути покажчиком. Результатом операції є величина, на яку вказує операнд. Типом результату є тип величини, що адресується покажчиком. Результат не визначений, якщо покажчик містить неприпустиму адресу.

Розглянемо типові ситуації, коли покажчик містить неприпустиму адресу:

- покажчик є нульовим;

- покажчик визначає адресу такого об'єкту, який не є активним у момент посилання;

- покажчик визначає адресу, яка не вирівняна до типу об'єкту, на який він вказує;

- покажчик визначає адресу, не використовувану програмою, що виконується.

Операція адреса (&) дає адресу свого операнда. Операндом може бути будь-яке іменоване вираження. Ім'я функції або масиву також може бути операндом операції "адреса", хоча в цьому випадку знак операції є зайвим, оскільки імена масивів і функцій є адресами. Результатом операції адреса є покажчик на операнд. Тип, що адресується покажчиком, є типом операнда.

Операція адреса не може застосуються до елементів структури, полями бітів, що являються, і до об'єктів з класом пам'яті register.

Приклади:

```
int t, f=0,  * adress;
adress = &t; /* змінній adress, що оголошується як
            покажчик, привласнюється адреса змінної t */
* adress =f; /* змінній що знаходиться за адресою, що міститься
            у змінній adress, привласнюється значення
            змінній f,  тобто  0, що еквівалентно
            t=f;      тобто   t=0;                  */
```

1.3.5. Операція sizeof

За допомогою операції sizeof можна визначити розмір пам'яті яка відповідає ідентифікатору або типу. Операція sizeof має наступний формат:

sizeof(вираження).

В якості вираження може бути використаний будь-який ідентифікатор, або ім'я типу, поміщене в дужки. Відмітимо, що не може бути використане ім'я типу void, а ідентифікатор не може відноситься до поля бітів або бути ім'ям функції.

Якщо в якості вираження вказане ім'я масиву, то результатом є розмір усього масиву (тобто твір числа елементів на довжину типу), а не розмір покажчика, що відповідає ідентифікатору масиву.

Коли sizeof застосовуються до імені типу структури або об'єднання або до ідентифікатора що має тип структури або об'єднання, то результатом є фактичний розмір структури або об'єднання, який може включати ділянки пам'яті, використовувані для вирівнювання елементів структури або об'єднання. Таким чином, цей результат може не відповідати розміру, що отримується шляхом складання розмірів елементів структури. Когда sizeof применяются к имени типа структуры или объединения или к идентификатору имеющему тип структуры или объединения, то результатом является фактический размер структуры или объединения, который может включать участки памяти, используемые для выравнивания элементов структуры или объе-

динения. Таким образом, этот результат может не соответствовать размеру, получаемому путем сложения размеров элементов структуры.

Приклад:

```
struct    { char   h;
            int    b;
            double f;
          } str;
int a1;
a1  = sizeof(str);
```

Змінна a1 отримає значення, рівне 12, в той же час якщо скласти довжини усіх використовуваних в структурі типів, то отримаємо, що довжина структури str дорівнює 7.

Невідповідність має місце на увазі того, що після розміщення в пам'яті першої змінної h довгої 1 байт, додається 1 байт для вирівнювання адреси змінною b на границю слова (слово має довжину 2 байти для машин серії IBM PC AT /286/287), далі здійснюється вирівнювання адреси змінною f на границю подвійного слова (4 байти), таким чином в результаті операцій вирівнювання для розміщення структури в оперативній пам'яті вимагається на 5 байт більше. Несоответствие имеет место в виду того, что после размещения в памяти первой переменной h длинной 1 байт, добавляется 1 байт для выравнивания адреса переменной b на границу слова (слово имеет длину 2 байта для машин серии IBM PC AT /286/287), далее осуществляется выравнивание адреса переменной f на границу двойного слова (4 байта), таким образом в результате операций выравнивания для размещения структуры в оперативной памяти требуется на 5 байт больше.

У зв'язку з цим доцільно рекомендувати при оголошенні структур і об'єднання розташовувати їх елементи в порядку убування довжини типів, тобто приведену вище структуру слід записати в наступному виді:

```
struct  { double f;
          int    b;
          char   h;
        } str;
```

1.3.6. Мультиплікативні операції

До цього класу операцій відносяться операції множення (*), ділення (/) і отримання залишку від ділення (%). Операндами операції (%) мають бути цілі числа. Відмітимо, що типи операндів операцій множення і ділення можуть відрізнятися, і для них справедливі правила перетворення типів. Типом результату є тип операндів після перетворення.

Операція множення (*) виконує множення операндів.

```
int  i=5;
float f=0.2;
double g, z;
   g=f*i;
```

Тип твору i і f перетвориться до типу double, потім результат привласнюється змінній g.

Операція ділення (/) виконує ділення першого операнда на другий. Якщо дві цілі величини не діляться без остачі, то результат округляється у бік нуля.

При спробі ділення на нуль видається повідомлення під час виконання.

```
int i=49, j=10,  n, m;
n = i/j;              /* результат   4   */
m = i/(- j);          /* результат  - 4   */
```

Операція залишок від ділення (%) дає залишок від ділення першого операнда на другий.

Знак результату залежить від конкретної реалізації. У цій реалізації знак результату співпадає зі знаком ділимого. Якщо другий операнд дорівнює нулю, то видається повідомлення.

```
int n = 49, m = 10, i, j, k, l;
i = n % m;            /*    9    */
j = n % (- m);        /*    9    */
k = (- n) % m;        /*    -9   */
l = (- n) %  (- m);   /*    -9   */
```

1.3.7. Аддитивні операції

До аддитивних операцій відносяться складання (+) і віднімання (-). Операнди можуть бути цілого або плаваючого типів. В деяких випадках над операндами аддитивних операцій виконуються загальні арифметичні перетворення. Проте перетворення, що виконуються при аддитивних операціях, не забезпечують обробку ситуацій переповнювання і втрати значущості. Інформація втрачається, якщо результат аддитивної операції не може бути представлений типом операндів після перетворення. При цьому повідомлення про помилку не видається. К аддитивным операциям относятся сложение (+) и вычитание (-). Операнды могут быть целого или плавающего типов. В некоторых случаях над операндами аддитивных операций выполняются общие арифметические преобразования. Однако преобразования, выполняемые при аддитивных операциях, не обеспечивают обработку ситуаций переполнения

и потери значимости. Информация теряется, если результат аддитивной операции не может быть представлен типом операндов после преобразования. При этом сообщение об ошибке не выдается.

Приклад:

```
int   i=30000, j=30000, k;
      k=i+j;
```

В результаті складання k отримає значення рівне - 5536.

Результатом виконання операції складання є сума двох операндів. Операнди можуть бути цілого або плаваючого типу або один операнд може бути покажчиком, а другий - цілою величиною.

Коли ціла величина складається з покажчиком, то ціла величина перетвориться шляхом множення її на розмір пам'яті, займаної величиною, що адресується покажчиком.

Коли перетворена ціла величина складається з величиною покажчика, то результатом є покажчик, що адресує елемент пам'яті, розташований на цілу величину далі від початкової адреси. Нове значення покажчика адресує той же самий тип даних, що і початковий покажчик.

Операція віднімання (-) віднімає другий операнд з першого. Можлива наступна комбінація операндів :

1. Обидва операнди цілого або плаваючого типу.

2. Обидва операнди є покажчиками на один і той же тип.

3. Перший операнд є покажчиком, а другий - цілим.

Відмітимо, що операції складання і віднімання над адресами в одиницях, відмінних від довжини типу, можуть привести до непередбачуваних результатів.

Приклад:

```
double d[10],* u;
int i;
u = d+2;  /*  u  вказує на третій елемент масиву */
i = u - d;  /*  i  набуває значення рівне 2         */
```

1.3.8. Операції зрушення

Операції зрушення здійснюють зміщення операнда вліво (<>) на число бітів, що задається другим операндом. Обидва операнди мають бути цілими величинами. Виконуються звичайні арифметичні перетворення. При зрушенні вліво праві біти, що звільняються, встановлюються в нуль. При зрушенні управо метод заповнення лівих бітів, що звільняються, залежить від типу першого операнда. Якщо тип unsigned, то вільні ліві біти встановлюються в нуль. Інакше вони заповнюються копією знакового біта. Результат операції зрушення не визначений, якщо другий операнд негативний. Операции сдвига осуществляют смещение операнда влево (<>) на число битов, задаваемое вторым операндом. Оба операнда должны быть целыми величинами. Выполняются обычные арифметические преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в нуль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа первого операнда. Если тип unsigned, то свободные левые биты устанавливаются в нуль. В противном случае они заполняются копией знакового бита. Результат операции сдвига не определен, если второй операнд отрицательный.

Перетворення, виконані операціями зрушення, не забезпечують обробку ситуацій переповнювання і втрати значущості. Інформація втрачається, якщо результат операції зрушення не може бути представлений типом першого операнда, після перетворення.

Відмітимо, що зрушення вліво відповідає множенню першого операнда на міру числа 2, рівну другому операнду, а зрушення управо відповідає діленню першого операнда на 2 в ступені, рівному другому операнду.

Приклади:

```
int  i=0x1234, j,  k ;
k = i<<>8 ;         /*    i = 0x0034    */
```

1.3.9. Порозрядні операції

До порозрядних операцій відносяться: операція порозрядного логічного "И" (&), операція порозрядного логічного "АБО" (|), операція порозрядного, що "виключає АБО" (^).

Операнди порозрядних операцій можуть бути будь-якого цілого типу. При необхідності над операндами виконуються перетворення за умовчанням, тип результату - це тип операндів після перетворення.

Операція порозрядного логічного І (&) порівнює кожен біт першого операнда з відповідним бітом другого операнда. Якщо обоє порівнюваних біта одиниці, то відповідний біт результату встановлюється в 1, інакше в 0.

Операція порозрядного логічного АБО (|) порівнює кожен біт першого операнда з відповідним бітом другого операнда. Якщо будь-який (чи обоє) з порівнюваних бітів дорівнює 1, то відповідний біт результату встановлюється в 1, інакше результуючий біт дорівнює 0.

Операція порозрядного, що виключає АБО (^) порівнює кожен біт першого операнда з відповідними бітами другого операнда. Якщо один з порівнюваних бітів дорівнює 0, а другий біт дорівнює 1, то відповідний біт результату встановлюється в 1, інакше, тобто коли обидва біти дорівнюють 1 або 0, біт результату встановлюється в 0.

Приклад.

```
int  i=0x45FF,     /*      i= 0100 0101 1111 1111   */
     j=0x00FF;             j= 0000 0000 1111 1111   */
char r;
  r = i^j;      /* r=0x4500 = 0100 0101 0000 0000   */
  r = i|j;      /* r=0x45FF = 0100 0101 0000 0000   */
  r = i&j;      /* r=0x00FF = 0000 0000 1111 1111    */
```

1.3.10. Логічні операції

До логічних операцій відносяться операція логічного І (&&) і операція логічного АБО (||). Операнди логічних операцій можуть бути цілого типу, плаваючого типу або типу покажчика, при цьому в кожній операції можуть брати участь операнди різних типів.

Операнди логічних виразів обчислюються зліва направо. Якщо значення першого операнда вистачає, щоб визначити результат операції, то другий операнд не обчислюється.

Логічні операції не викликають стандартних арифметичних перетворень. Вони оцінюють кожен операнд з точки зору його еквівалентності нулю. Результатом логічної операції є 0 або 1, тип результату int.

Операція логічного І (&&) виробляє значення 1, якщо обидва операнди мають нульові значення. Якщо один з операндів дорівнює 0, то результат також дорівнює 0. Якщо значення першого операнда дорівнює 0, то другий операнд не обчислюється.

Операція логічного АБО (||) виконує над операндами операцію того, що включає АБО. Вона виробляє значення 0, якщо обидва операнди мають значення 0, якщо який-небудь з операндів має ненульове значення, то результат операції дорівнює 1. Якщо перший операнд має ненульове значення, то другий операнд не обчислюється.

1.3.11. Операція послідовного обчислення

Операція послідовного обчислення позначається комі (,) і використовується для обчислення двох і більше виразів там, де по синтаксису допустимо тільки одне вираження. Ця операція обчислює два операнди зліва направо. При виконанні операції послідовного обчислення, перетворення типів не робиться. Операнди можуть бути будь-яких типів. Результат операції має значення і тип другого операнда. Відмітимо, що кома може використовуватися також як символ роздільник, тому необхідно по контексту розрізняти, кому, що використовується як роздільник або знак операції. Операция последовательного вычисления обозначается запятой (,) и используется для вычисления двух и более выражений там, где по синтаксису допустимо только одно выражение. Эта операция вычисляет два операнда слева направо. При выполнении операции последовательного вычисления, преобразование типов не производится. Операнды могут быть любых типов. Результат операции имеет значения и тип второго операнда. Отметим, что запятая может использоваться также как символ разделитель, поэтому необходимо по контексту различать, запятую, используемую в качестве разделителя или знака операции.

1.3.12. Умовна операція

У мові CI є одна тернарная операція - умовна операція, яка має наступний формат :

операнд-1 ? операнд-2: операнд-3

Операнд-1 має бути цілого або плаваючого типу або бути покажчиком. Він оцінюється з точки зору його еквівалентності 0. Якщо операнд-1 не рівний 0, то обчислюється операнд-2 і його значення є результатом операції. Якщо операнд-1 рівний 0, то обчислюється операнд-3 і його значення є результатом операції. Слід зазначити, що обчислюється або операнд-2, або операнд-3, але не обоє. Тип результату залежить від типів операнда-2 і операнда-3, таким чином.

1. Якщо операнд-2 або операнд-3 має цілий або плаваючий тип (відмітимо, що їх типи можуть відрізнятися), то виконуються звичайні арифметичні перетворення. Типом результату є тип операнда після перетворення.

2. Якщо операнд-2 і операнд-3 мають один і той же тип структури, об'єднання або покажчика, то тип результату буде тим же самим типом структури, об'єднання або покажчика.

3. Якщо обидва операнди мають тип void, то результат має тип void.

4. Якщо один операнд є покажчиком на об'єкт будь-якого типу, а інший операнд є покажчиком на vold, то покажчик на об'єкт перетвориться до покажчика на vold, який і буде типом результату.

5. Якщо один з операндів є покажчиком, а інший константним вираженням зі значенням 0, то типом результату буде тип покажчика.

Приклад:

max = (d<=b)? b: d;

Змінній max привласнюється максимальне значення змінних d і b.

1.3.13. Операції збільшення і зменшення

Операції збільшення (++) і зменшення (--) є унарними операціями привласнення. Вони відповідно збільшують або зменшують значення операнда на одиницю. Операнд може бути цілого або плаваючого типу або типу покажчик і має бути таким, що модифікується. Операнд цілого або плаваючого типу збільшуються (зменшуються) на одиницю. Тип результату відповідає типу операнда. Операнд адресного типу збільшується або зменшується на розмір об'єкту, який він адресує. У мові допускається префіксна або постфіксна форми операцій збільшення (зменшення), тому значення вираження, що використовує операції збільшення (зменшення) залежить від того, яка з форм вказаних операцій використовується. Операции увеличения (++) и уменьшения (--) являются унарными операциями присваивания. Они соответственно увеличивают или уменьшают значения операнда на единицу. Операнд может быть целого или плавающего типа или типа указатель и должен быть модифицируемым. Операнд целого или плавающего типа увеличиваются (уменьшаются) на единицу. Тип результата соответствует типу операнда. Операнд адресного типа увеличивается или уменьшается на размер объекта, который он адресует. В языке допускается префиксная или постфиксная формы операций увеличения (уменьшения), поэтому значения выражения, использующего операции увеличения (уменьшения) зависит от того, какая из форм указанных операций используется.

Якщо знак операції стоїть перед операндом (префіксна форма запису), то зміна операнда відбувається до його використання у вираженні і результатом операції є збільшене або зменшене значення операнда.

У тому разі якщо знак операції стоїть після операнда (постфіксна форма запису), то операнд спочатку використовується для обчислення вираження, а потім відбувається зміна операнда.

Приклади:

```
int t=1, s=2, z, f;
    z=(t++)*5;
```

Спочатку відбувається множення t*5, а потім збільшення t. В результаті вийде z=5, t=2.

```
f=(++s)/3;
```

Спочатку значення s збільшується, а потім використовується в операції ділення. В результаті отримаємо s=3, f=1.

У разі, якщо операції збільшення і зменшення використовуються як самостійні оператори, префіксна і постфіксна форми запису стають еквівалентними.

```
z++; /* еквівалентно */ ++z;
```

1.3.14. Просте привласнення

Операція простого привласнення використовується для заміни значення лівого операнда, значенням правого операнда. При привласненні робиться перетворення типу правого операнда до типу лівого операнда за правилами, згаданими раніше. Лівий операнд має бути таким, що модифікується.

Приклад:

```
int t;
char f;
long z;
t=f+z;
```

Значення змінної f перетвориться до типу long, обчислюється f+z, результат перетвориться до типу int і потім привласнюється змінній t.

1.3.15. Складене привласнення

Окрім простого привласнення, є ціла група операцій привласнення, які об'єднують просте привласнення з однією з бінарних операцій. Такі операції називаються складеними операціями привласнення і мають вигляд:

(операнд-1) (бінарна операція) = (операнд-2).

Складене привласнення по результату еквівалентно наступному простому привласненню:

(операнд-1) = (операнд-1) (бінарне операція) (операнд-2).

Відмітимо, що вираження складеного привласнення з точки зору реалізації не еквівалентне простому привласненню, оскільки в останньому операнд-1 обчислюється двічі.

Кожна операція складеного привласнення виконує перетворення, які здійснюються відповідною бінарною операцією. Лівим операндом операцій (+=) (-=) може бути покажчик, тоді як правий операнд має бути цілим числом.

Приклади:

```
double  arr[4]={ 2.0, 3.3, 5.2, 7.5 } ;
double  b=3.0;
b+=arr[2];     /* еквівалентно   b=b+arr[2]          */
arr[3]/=b+1;   /* еквівалентно   arr[3]=arr[3]/(b+1)   */
```

Помітимо, що при другому привласненні використання складеного привласнення дає помітніший виграш в часі виконання, оскільки лівий операнд є індексним вираженням.

1.3.16. Пріоритети операцій і порядок обчислень

У мові СІ операції з вищими пріоритетами обчислюються першими. Найвищим пріоритетом є пріоритет рівний 1. Пріоритети і порядок операцій приведені в таблицю. 8.

Таблиця 8

Пріоритет	**Знак операції**	**Типи операції**	**Порядок виконання**
2	() [] . ->	Вираження	Зліва направо
1	- ~ ! * & ++ -- sizeof приведення типів	Унарні	Справа наліво
3	* / %	Мультиплікативні	Зліва направо
4	+ -	Адитивні	
5	<< >>	Зрушення	
6	< > <= >=	Відношення	
7	== !=	Відношення (рівність)	
8	&	Порозрядне І	
9	^	Що порозрядне, що виключає АБО	
10	|	Порозрядне АБО	

11	&&	Логічне І	
12	||	Логічне АБО	
13	? :	Умовна	
14	= *= /= %= += -= &= |= >>= <<= ^=	Просте і складене привласнення	Справа наліво
15	,	Послідовне обчислення	Зліва направо

1.3.17. Побічні ефекти

Операції привласнення в складних виразах можуть викликати побічні ефекти, оскільки вони змінюють значення змінної. Побічний ефект може виникати і при виклику функції, якщо він містить пряме або непряме привласнення (через покажчик). Це пов'язано з тим, що аргументи функції можуть обчислюватися у будь-якому порядку. Наприклад, побічний ефект має місце в наступному виклику функції :

prog (a, a=k*2);

Залежно від того, який аргумент обчислюється першим, у функцію можуть бути передані різні значення.

Порядок обчислення операндів деяких операцій залежить від реалізації і тому можуть виникати різні побічні ефекти, якщо в одному з операндів використовується операції збільшення або зменшення, а також інші операції привласнення.

Наприклад, вираження i*j+(j++)+ (--i) може набувати різних значень при обробці різними компіляторами. Щоб уникнути непорозумінь при виконанні побічних ефектів необхідно дотримуватися наступних правил.

1. Не використати операції привласнення змінної у виклику функції, якщо ця змінна бере участь у формуванні інших аргументів функції.

2. Не використати операції привласнення змінною у вираженні, якщо ця змінна використовується у вираженні більше одного разу.

1.3.18. Перетворення типів

При виконанні операцій відбуваються неявні перетворення типів в наступних випадках:

- при виконанні операцій здійснюються звичайні арифметичні перетворення (які були розглянуті вище);

- при виконанні операцій привласнення, якщо значення одного типу привласнюється змінній іншого типу;

- при передачі аргументів функції.

Крім того, в Сі є можливість явного приведення значення одного типу до іншого.

У операціях привласнення тип значення, яке привласнюється, перетвориться до типу змінної, що набуває цього значення. Допускається перетворення цілих і плаваючих типів, навіть якщо таке перетворення веде до втрати інформації.

Перетворення цілих типів зі знаком. Ціле зі знаком перетвориться до короткого цілого зі знаком, за допомогою усікання старших бітів. Ціла зі знаком перетвориться до довшого цілого зі знаком, шляхом розмноження знаку. При перетворенні цілого зі знаком до цілого без знаку, ціле зі знаком перетвориться до розміру цілого без знаку і результат розглядається як значення без знаку.

Перетворення цілого зі знаком до плаваючого типу відбувається без втрати] інформації, за винятком випадку перетворення значення типу long int або unsigned long int до типу float, коли точність часто може бути втрачена.

Перетворення цілих типів без знаку. Ціле без знаку перетвориться до короткого цілого без знаку або зі знаком шляхом усікання старших бітів. Ціле без знаку перетвориться до довшого цілого без знаку або зі знаком шляхом доповнення нулів ліворуч.

Коли ціле без знаку перетвориться до цілого зі знаком того ж розміру, бітове представлення не змінюється. Тому значення, яке воно представляє, змінюється, якщо знаковий біт встановлений (рівний 1), тобто коли початкове ціле без знаку більше ніж максимальне позитивне ціле зі знаком, такої ж довжини.

Цілі значення без знаку перетворяться до плаваючого типу, шляхом перетворення цілого без знаку до значення типу signed long, а потім значення signed long перетвориться в плаваючий тип. Перетворення з unsigned long до типу float, double або long double робляться з втратою інформації, якщо перетворюване значення більше, ніж максимальне позитивне значення, яке може бути представлене для типу long.

Перетворення плаваючих типів. Величини типу float перетворяться до типу double без зміни значення. Величини double і long double перетворяться до float с деякою втратою точності. Якщо значення занадто велике для float, то відбувається втрата значущості, про що повідомляється під час виконання.

При перетворенні величини з плаваючою точкою до цілих типів вона спочатку перетвориться до типу long (дробова частина плаваючої величини при цьому відкидається), а потім величина типу long перетвориться до необхідного цілого типу. Якщо значення занадто велике для long, то результат перетворення не визначений.

Перетворення з float, double або long double до типу unsigned long робиться з втратою точності, якщо перетворюване значення більше, ніж максимально можливе позитивне значення, представлене типом long.

Перетворення типів покажчика. Покажчик на величину одного типу може бути перетворений до покажчика на величину іншого типу. Проте результат може бути не визначений із-за відмінностей у вимогах до вирівнювання і розмірах для різних типів.

Покажчик на тип void може бути перетворений до покажчика на будь-який тип, і покажчик на будь-який тип може бути перетворений до покажчика на тип void без обмежень. Значення покажчика може бути перетворене до цілої величини. Метод перетворення залежить від розміру покажчика і розміру цілого типу таким чином:

- якщо розмір покажчика менше розміру цілого типу або дорівнює йому, то покажчик перетвориться точно так, як і ціле без знаку;

- якщо покажчик більший, ніж розмір цілого типу, то покажчик спочатку перетвориться до покажчика з тим же розміром, що і цілий тип, і потім перетвориться до цілого типу.

Цілий тип може бути перетворений до адресного типу за наступними правилами:

- якщо цілий тип того ж розміру, що і покажчик, то ціла величина просто розглядається як покажчик (ціле без знаку);

- якщо розмір цілого типу відмінний від розміру покажчика, то цілий тип спочатку перетвориться до розміру покажчика (використовуються способи перетворення, описані вище), а потім отримане значення трактується як покажчик.

Перетворення при виклику функції. Перетворення, що виконуються над аргументами при виклику функції, залежать від того, чи був заданий прототип функції (оголошення "вперед") зі списком оголошень типів аргументів.

Якщо заданий прототип функції і він включає оголошення типів аргументів, то над аргументами у виклику функції виконуються тільки звичайні арифметичні перетворення.

Ці перетворення виконуються незалежно для кожного аргументу. Величини типу float перетворяться до double, величини типу char і short перетворяться до int, величини типів unsigned char і unsigned short перетворяться до unsigned int. Можуть бути також виконані неявні перетворення змінних типу покажчик. Задаючи прототипи функцій, можна перевизначити ці неявні перетворення і дозволити компілятору виконати контроль типів.

Перетворення при приведенні типів. Явне перетворення типів може бути здійснене за допомогою операції приведення типів, яка має формат:

(имя-типа) операнд .

У приведеному записі имя-типа задає тип, до якого має бути перетворений операнд.

Приклад:

```
int     i=2;
long    l=2;
double    d;
float     f;
d=(double) i * (double) l;
f=(float) d;
```

У цьому прикладі величини i, l, d явно перетворюватимуться до вказаних в круглих дужках типів.

1.4. Оператори

Усі оператори мови СІ можуть бути умовно розділені на наступні категорії:

- умовні оператори, до яких відносяться оператор умови if і оператор вибору switch;

- оператори циклу (for, while, do while);

- оператори переходу (break, continue, return, goto);

- інші оператори (оператор "вираження", порожній оператор).

Оператори в програмі можуть об'єднуватися в складені оператори за допомогою фігурних дужок. Будь-який оператор в програмі може бути помічений міткою, що складається з імені і двокрапки, що йде за ним.

Усі оператори мови СІ, окрім складених операторів, закінчуються крапкою з комою ";".

1.4.1. Оператор вираження

Будь-яке вираження, яке закінчується крапкою з комою, є оператором.

Виконання оператора вираження полягає в обчисленні вираження. Отримане значення вираження ніяк не використовується, тому, як правило, такі вирази викликають побічні ефекти. Помітимо, що викликати функцію, що не повертає значення можна тільки за допомогою оператора вираження. Правила обчислення виразів були сформульовані вище.

Приклади:

```
++ i;
```

Цей оператор представляє вираження, яке збільшує значення змінної і на одиницю.

```
a=cos(b * 5);
```

Цей оператор представляє вираження, що включає операції привласнення і виклику функції.

```
a(x, y);
```

Цей оператор представляє вираження що складається з виклику функції.

1.4.2. Порожній оператор

Порожній оператор складається тільки з крапки з комою. При виконанні цього оператора нічого не відбувається. Він зазвичай використовується в наступних випадках:

- у операторах do, for, while, if в рядках, коли місце оператора не потрібно, але по синтаксису потрібно хоч би одного оператора;

- при необхідності помітити фігурну дужку.

Синтаксис мови СІ вимагає, щоб після мітки обов'язково слідував оператор. Фігурна ж дужка оператором не є. Тому, якщо потрібно передати управління на фігурну дужку, необхідно використати порожній оператор.

Приклад:

```
int main ( )
```

```
{
   :
    { if (...) goto a;    /* перехід на дужку */
       { ...
       }
a:;  }
    return 0;
 }
```

1.4.3. Складений оператор

Складений оператор є декількома операторами і оголошеннями, поміщеними у фігурні дужки :

```
{  [объявление]
   :
   оператор; [оператор];
   :
}
```

Помітимо, що у кінці складеного оператора крапка з комою не ставиться.

Виконання складеного оператора полягає в послідовному виконанні складових його операторів.

Приклад:

```
int main ()
{
  int    q, b;
  double t, d;
    :
    if (...)
      {
        int    e, g;
        double f, q;
        :
       }
    :
    return (0);
}
```

Змінні e, g, f, q будуть знищені після виконання складеного оператора. Відмітимо, що змінна q є локальною в складеному операторові, тобто вона жодним чином не пов'язана зі змінною q оголошеної на початку функції main з типом

int. Відмітимо також, що вираження стоїть після return може бути поміщено в круглі дужки, хоча наявність останніх необов'язково.

1.4.4. Оператор if

Формат оператора :

if (вираження) оператор-1; [else оператор-2;]

Виконання оператора if розпочинається з обчислення вираження.

Далі виконання здійснюється за наступною схемою:

- якщо вираження істинне (тобто відмінно від 0), то виконується оператор-1.

- якщо вираження неправдиве (тобто рівне 0), то виконується оператор-2.

- якщо вираження неправдиво і відсутнє оператор-2 (у квадратні дужки поміщена необов'язкова конструкція), то виконується той, що йде за if оператор.

Після виконання оператора if значення передається на наступний оператор програми, якщо послідовність виконання операторів програми не буде примусово порушена використанням операторів переходу.

Приклад:

```
if (i < j)   i++:
else { j = i - 3;   i++; }
```

Цей приклад ілюструє також і той факт, що на місці оператор-1, так само як і на місці оператор-2 можуть знаходитися складні конструкції.

Допускається використання вкладених операторів if. Оператор if може бути включений в конструкцію if або в конструкцію else іншого оператора if. Щоб зробити програму більше читабельною, рекомендується групувати оператори і конструкції у вкладених операторах if, використовуючи фігурні дужки. Якщо ж фігурні дужки опущені, то компілятор зв'язує кожне ключове слово else з найбільш близьким if, для якого немає else.

Приклади:

```
int main ( )
 {
  int t=2, b=7, r=3;
    if (t>b)
     {
        if (b < r)  r=b;
```

```
    }
   else r=t;
  return (0);
}
```

В результаті виконання цієї програми r стане рівним 2.

Якщо ж в програмі опустити фігурні дужки, що стоять після оператора if, то програма матиме наступний вигляд:

```
int main  ( )
 {
  int t=2, b=7, r=3;
  if ( a>b )
     if ( b < c ) t=b;
     else       r=t;
  return (0);
 }
```

В цьому випадку r отримає значення рівне 3, оскільки ключове слово else відноситься до другого оператора if, який не виконується, оскільки не виконується умова, що перевіряється в першому операторові if.

Наступний фрагмент ілюструє вкладені оператори if :

```
char ZNAC;
int x, y, z;
  :
 if (ZNAC == '-') x = y - z;
 else  if (ZNAC == '+') x = y + z;
       else if (ZNAC == '*') x = y * z;
            else if (ZNAC == '/') x = y / z;
                 else ...
```

З розгляду цього прикладу можна зробити висновок, що конструкції використовуючі вкладені оператори if, є досить громіздкими і не завжди досить надійними. Іншим способом організації вибору з безлічі різних варіантів являється використання спеціального оператора вибору switch.

1.4.5. Оператор switch

Оператор switch призначений для організації вибору з безлічі різних варіантів. Формат оператора наступний:

```
switch ( вираження )
{   [оголошення]
      :
```

```
    [ case  константное-выражение1]: [ список-операторов1]
    [ case  константное-выражение2]: [ список-операторов2]
      :
      :
    [ default: [ список операторів ]]
}
```

Вираження, що йде за ключовим словом switch в круглих дужках, може бути будь-яким вираженням, допустимими в мові CI, значення якого має бути цілим. Відмітимо, що можна використати явне приведення до цілого типу, проте необхідно пам'ятати про ті обмеження і рекомендації, про які говорилося вище.

Значення цього виразу є ключовим для вибору з декількох варіантів. Тіло оператора smitch складається з декількох операторів, помічених ключовим словом case з подальшим константным-выражением. Слід зазначити, що використання цілого константного вираження є істотним недоліком, властивим розглянутому операторові.

Оскільки константне вираження обчислюється під час трансляції, воно не може містити змінні або виклики функцій. Зазвичай як константне вираження використовуються цілі або символьні константи.

Усі константні вирази в операторові switch мають бути унікальні. Окрім операторів, помічених ключовим словом case, можливо, але обов'язково один, фрагмент помічений ключовим словом default.

Список операторів може бути порожнім, або містити один або більше за операторів. Причому в операторові switch не вимагається брати послідовність операторів у фігурних дужок.

Відмітимо також, що в операторові switch можна використати свої локальні змінні, оголошення яких знаходяться перед першим ключовим словом case, проте в оголошеннях не повинна використовуватися ініціалізація.

Схема виконання оператора switch наступна:

- обчислюється вираження в круглих дужках;

- вичислені значення послідовно порівнюються з константними виразами, що йдуть за ключовими словами case;

- якщо один з константних виразів співпадає зі значенням вираження, то управління передається на оператор, помічений відповідним ключовим словом case;

- якщо жоден з константних виразів не дорівнює вираженню, то управління передається на оператор, помічений ключовим словом default, а у разі його відсутності управління передається на наступний після switch оператор.

Відмітимо цікаву особливість використання оператора switch : конструкція із словом default може бути не останньою в тілі оператора switch. Ключові слова case і default в тілі оператора switch істотні тільки при початковій перевірці, коли визначається початкова точка виконання тіла оператора switch. Усі оператори, між початковим оператором і кінцем тіла, виконуються незалежно від ключових слів, якщо тільки якийсь з операторів не передасть управління з тіла оператора switch. Таким чином, програміст повинен сам потурбується про вихід з case, якщо це необхідно. Найчастіше для цього використовується оператор break. Отметим интересную особенность использования оператора switch: конструкция со словом default может быть не последней в теле оператора switch. Ключевые слова case и default в теле оператора switch существенны только при начальной проверке, когда определяется начальная точка выполнения тела оператора switch. Все операторы, между начальным оператором и концом тела, выполняются вне зависимости от ключевых слов, если только какой-то из операторов не передаст управления из тела оператора switch. Таким образом, программист должен сам позаботится о выходе из case, если это необходимо. Чаще всего для этого используется оператор break.

Для того, щоб виконати одні і ті ж дії для різних значень вираження, можна помітити один і той же оператор декількома ключовими словами case.

Приклад:

```
int i=2;
switch (i)
  {
   case 1: i += 2;
   case 2: i *= 3;
   case 0: i /= 2;
   case 4: i -= 5;
   default:    ;
}
```

Виконання оператора switch розпочинається з оператора, поміченого case 2. Таким чином, змінна і набуває значення, рівного 6, далі виконується оператор, помічений ключовим словом case 0, а потім case 4, змінна і набуде значення 3, а потім значення - 2. Оператор, помічений ключовим словом default, не змінює значення змінної.

Розглянемо раніше наведений приклад, в якому ілюструвалося використання вкладених операторів if, переписаною тепер з використанням оператора switch.

```
char ZNAC;
int x, y, z;
```

```
switch (ZNAC)
{
    case '+':  x = y + z;    break;
    case '-':  x = y - z;    break;
    case '*':  x = y * z;    break;
    case '/':  x = u / z;    break;
    default: ;
  }
```

Використання оператора break дозволяє в необхідний момент перервати послідовність виконуваних операторів в тілі оператора switch, шляхом передачі управління операторові, що йде за switch.

Відмітимо, що в тілі оператора switch можна використати вкладені оператори switch, при цьому в ключових словах case можна використати однакові константні вирази.

Приклад:

```
  :
switch (a)
 {
  case 1: b=c; break;
  case 2:
     switch (d)
     {   case 0:  f=s;  break;
         case 1:  f=9;  break;
         case 2:  f-=9; break;
     }
   case 3: b-=c; break;
    :
 }
```

1.4.6. Оператор break

Оператор break забезпечує припинення виконання самого внутрішнього з операторів switch, що об'єднують його, do, for, while. Після виконання оператора break управління передається операторові, що йде за перерваним.

1.4.7. Оператор for

Оператор for - це найбільш загальний спосіб організації циклу. Він має наступний формат:

for (вираження 1 ; вираження 2 ; вираження 3) тіло

Вираження 1 зазвичай використовується для встановлення початкового значення змінних, що управляють циклом. Вираження 2 - цей вираз, що визначає умову, при якій тіло циклу виконуватиметься. Вираження 3 визначає зміну змінних, що управляють циклом після кожного виконання тіла циклу.

Схема виконання оператора for :

1. Обчислюється вираження 1.

2. Обчислюється вираження 2.

3. Якщо значення вираження 2 відмінно від нуля (істина), виконується тіло циклу, обчислюється вираження 3 і здійснюється перехід до пункту 2, якщо вираження 2 дорівнює нулю (брехня), то управління передається на оператор, що йде за оператором for.

Істотне те, що перевірка умови завжди виконується на початку циклу. Це означає, що тіло циклу може жодного разу не виконатися, якщо умова виконання відразу буде неправдивою.

Приклад:

```
int main()
{ int i, b;
  for (i=1; i
```

В даному прикладі обчислюються квадрати чисел від 1 до 9.

Деякі варіанти використання оператора for підвищують його гнучкість за рахунок можливості використання декількох змінних, що управляють циклом.

Приклад:

```
int main()
{  int top,  bot;
   char string[100],  temp;
   for ( top=0, bot=100 ; top < bot ; top++, bot--)
   {  temp=string[top];
      string[bot]=temp;
   }
   return 0;
 }
```

В даному прикладі, що реалізовує запис рядка символів в зворотному порядку, для управління циклом використовуються дві змінні top і bot. Відмітимо,

що на місці вираження 1 і вираження 3 тут використовуються декілька виразів, записаних через кому, і виконуваних послідовно.

Іншим варіантом використання оператора for є нескінченний цикл. Для організації такого циклу можна використати порожнє умовне вираження, а для виходу з циклу зазвичай використовують додаткова умова і оператор break.

Приклад:

```
for (;;)
{ ...
  ...  break;
  ...
}
```

Оскільки згідно з синтаксисом мови Сі оператор може бути порожнім, тіло оператора for також може бути порожнім. Така форма оператора може бути використана для організації пошуку.

Приклад:

for (i=0; t[i]

У цьому прикладі змінна циклу і набуває значення номера першого елементу масиву t, значення якого більше 10.

1.4.8. Оператор while

Оператор циклу while називається циклом з передумовою і має наступний формат:

while (вираження) тіло ;

В якості вираження допускається використати будь-яке вираження мови Сі, а в якості тіла будь-який оператор, у тому числі порожній або складений. Схема виконання оператора while наступна:

1. Обчислюється вираження.

2. Якщо вираження неправдиве, то виконання оператора while закінчується і виконується наступний по порядку оператор. Якщо вираження істинне, то виконується тіло оператора while.

3. Процес повторюється з пункту 1.

Оператор циклу виду

for (вираження-1; вираження-2; вираження-3) тіло ;

може бути замінений оператором while таким чином:

```
вираження-1;
while (вираження-2)
{  тіло
   вираження-3;
}
```

Так само як і при виконанні оператора for, в операторові while спочатку відбувається перевірка умови. Тому оператор while зручно використати в ситуаціях, коли тіло оператора не завжди треба виконувати.

Усередині операторів for і while можна використати локальні змінні, які мають бути оголошені з визначенням відповідних типів.

1.4.9. Оператор do while

Оператор циклу do while називається оператором циклу з постумовою і використовується в тих випадках, коли необхідно виконати тіло циклу хоч би один раз. Формат оператора має наступний вигляд:

do тіло while (вираження);

Схема виконання оператора do while :

1. Виконується тіло циклу (яке може бути складеним оператором).

2. Обчислюється вираження.

3. Якщо вираження неправдиве, то виконання оператора do while закінчується і виконується наступний по порядку оператор. Якщо вираження істинне, то виконання оператора триває з пункту 1.

Щоб перервати виконання циклу до того, як умова стане неправдивою, можна використати оператор break.

Оператори while і do while можуть бути вкладеними.

Приклад:

```
int i, j, k;
...
i=0;  j=0;  k=0;
do { i++;
     j--;
     while (a[k] < i)  k++;
    }
```

```
while (i<- 30);
```

1.4.10. Оператор continue

Оператор continue, як і оператор break, використовується тільки усередині операторів циклу, але на відміну від нього виконання програми триває не з оператора, що йде за перерваним оператором, а з початку перерваного оператора. Формат оператора наступний:

continue;

Приклад:

```
int main()
{   int a, b;
   for (a=1, b=0; a
```

Коли сума чисел від 1 до a стає непарною, оператор continue передає управління на чергову ітерацію циклу for, не виконуючи оператори обробки парних сум.

Оператор continue, як і оператор break, перериває самий внутрішній з охоплюючих його циклів.

1.4.11. Оператор return

Оператор return завершує виконання функції, в якій він заданий, і повертає управління в зухвалу функцію, в точку, що безпосередньо йде за викликом. Функція main передає управління операційній системі. Формат оператора :

return [вираження] ;

Значення вираження, якщо воно задане, повертається в зухвалу функцію в якості значення функції, що викликається. Якщо вираження опущене, то повертане значення не визначене. Вираження може бути поміщене в круглі дужки, хоча їх наявність не обов'язкова.

Якщо в якій-небудь функції відсутній оператор return, то передача управління в зухвалу функцію відбувається після виконання останнього оператора функції, що викликається. При цьому повертане значення не визначене. Якщо функція не повинна мати повертаного значення, то її треба оголошувати з типом void.

Таким чином, використання оператора return потрібне або для негайного виходу з функції, або для передачі повертаного значення.

Приклад:

```
int sum (int a, int b)
{   renurn (a+b);    }
```

Функція sum має два формальні параметри a і b типу int, і повертає значення типу int, про що говорить описувач, що стоїть перед ім'ям функції. Повертане оператором return значення дорівнює сумі фактичних параметрів.

Приклад:

```
void prov (int a, double b)
{  double c;
  if (a10) return;
      else { c=a+b;
            if ((2*c - b)==11) return;
           }
}
```

В даному прикладі оператор return використовується для виходу з функції у разі виконання однієї з умов, що перевіряються.

1.4.12. Оператор goto

Використання оператора безумовного переходу goto в практиці програмування на мові CI настійно не рекомендується, оскільки він утрудняє розуміння програм і можливість їх модифікацій.

Формат цього оператора наступний:

```
goto имя-метки;
        ...
имя-метки: оператор;
```

Оператор goto передає управління на оператор, помічений міткою имя-метки. Помічений оператор повинен знаходитися в тій же функції, що і оператор goto, а використовувана мітка має бути унікальною, тобто одне имя-метки не може бути використане для різних операторів програми. Имя-метки - це ідентифікатор.

Будь-який оператор в складеному операторові може мати свою мітку. Використовуючи оператор goto, можна передавати управління всередину складеного оператора. Але треба бути обережним при вході в складений оператор, що містить оголошення змінних з ініціалізацією, оскільки оголошення розташовуються перед виконуваними операторами і значення оголошених змінних при такому переході будуть не визначені.
Любой оператор в составном операторе может иметь свою метку. Используя оператор goto, можно передавать управление внутрь составного оператора. Но нужно быть осторожным при входе в составной оператор, содержащий объявления переменных с инициализацией, так как объявления располагают-

ся перед выполняемыми операторами и значения объявленных переменных при таком переходе будут не определены.

1.5. ФУНКЦІЇ

1.5.1. Визначення і виклик функцій

Потужність мови СІ багато в чому визначається легкістю і гнучкістю у визначенні і використанні функцій в Сі-програмах. На відміну від інших мов програмування високого рівня в мові СІ немає ділення на процедури, підпрограми і функції, тут уся програма будується тільки з функцій.

Функція - це сукупність оголошень і операторів, зазвичай призначена для вирішення певного завдання. Кожна функція повинна мати ім'я, яке використовується для її оголошення, визначення і виклику. У будь-якій програмі на СІ має бути функція з ім'ям main (головна функція), саме з цієї функції, в якому б місці програми вона не знаходилася, починається виконання програми. Функция - это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе на СИ должна быть функция с именем main (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

При виклику функції їй за допомогою аргументів (формальних параметрів) можуть бути передані деякі значення (фактичні параметри), використовувані під час виконання функції. Функція може повертати деяке (одне !) значення. Це повертане значення і є результат виконання функції, який при виконанні програми підставляється в точку виклику функції, де б цей виклик не зустрівся. Допускається також використати функції що не мають аргументів і функції що не повертають ніяких значень. Дія таких функцій може полягати, наприклад, в зміні значень деяких змінних, виводі на друк деяких текстів і т.п. При вызове функции ей при помощи аргументов (формальных параметров) могут быть переданы некоторые значения (фактические параметры), используемые во время выполнения функции. Функция может возвращать некоторое (одно !) значение. Это возвращаемое значение и есть результат выполнения функции, который при выполнении программы подставляется в точку вызова функции, где бы этот вызов ни встретился. Допускается также использовать функции не имеющие аргументов и функции не возвращающие никаких значений. Действие таких функций может состоять, например, в изменении значений некоторых переменных, выводе на печать некоторых текстов и т.п..

З використанням функцій в мові СІ пов'язано три поняття - визначення функції (опис дій, що виконуються функцією), оголошення функції (завдання форми звернення до функції) і виклик функції.

Визначення функції задає тип повертаного значення, ім'я функції, типи і число формальних параметрів, а також оголошення змінних і оператори, що називаються тілом функції, і визначальні дію функції. У визначенні функції також може бути заданий клас пам'яті.

Приклад:

```
int rus (unsigned char r)
{  if (r>='A' && c<=' ')
     return 1;
   else
     return 0;
}
```

У цьому прикладі визначена функція з ім'ям rus, що має один параметр з ім'ям r і типом unsigned char. Функція повертає ціле значення, рівне 1, якщо параметр функції є буквою російського алфавіту, або 0 інакше.

У мові СІ немає вимоги, щоб визначення функції обов'язково передувало її виклику. Визначення використовуваних функцій можуть йти за визначенням функції main, перед ним, або знаходиться в іншому файлі.

Проте для того, щоб компілятор міг здійснити перевірку відповідності типів передаваних фактичних параметрів типам формальних параметрів до виклику функції треба помістити оголошення (прототип) функції.

Оголошення функції має такий же вигляд, що і визначення функції, з тією лише різницею, що тіло функції відсутнє, і імена формальних параметрів теж можуть бути опущені. Для функції, визначеної в останньому прикладі, прототип може мати вигляд

int rus (unsigned char r); чи rus (unsigned char);

У програмах на мові СІ широко використовуються, так звані, бібліотечні функції, тобто функції заздалегідь розроблені і записані у бібліотеки. Прототипи бібліотечних функцій знаходяться в спеціальних заголовних файлах, що поставляються разом з бібліотеками у складі систем програмування, і включаються в програму за допомогою директиви #include. В программах на языке СИ широко используются, так называемые, библиотечные функции, т.е. функции предварительно разработанные и записанные в библиотеки. Прототипы библиотечных функций находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования, и включаются в программу с помощью директивы #include.

Якщо оголошення функції не задане, то за умовчанням будується прототип функції на основі аналізу першого посилання на функцію, будь то виклик функції або визначення. Проте такий прототип не завжди узгоджується з подальшим визначенням або викликом функції. Рекомендується завжди задавати прототип функції. Це дозволить компілятору або видавати діагностичні повідомлення, при неправильному використанні функції, або коректним чином регулювати невідповідність аргументів встановлюване при виконанні програми. Если объявление функции не задано, то по умолчанию строится прототип функции на основе анализа первой ссылки на функцию, будь то вызов функ-

ции или определение. Однако такой прототип не всегда согласуется с последующим определением или вызовом функции. Рекомендуется всегда задавать прототип функции. Это позволит компилятору либо выдавать диагностические сообщения, при неправильном использовании функции, либо корректным образом регулировать несоответствие аргументов устанавливаемое при выполнении программы.

Оголошення параметрів функції при її визначенні може бути виконане в так званому "старому стилі", при якому в дужках після імені функції слідують тільки імена параметрів, а після дужок оголошення типів параметрів. Наприклад, функція rus з попереднього прикладу може бути визначена таким чином:

```
int rus (r)
unsigned char r;
{ ...    /*  тіло функції   */    ... }
```

Відповідно до синтаксису мови CI визначення функції має наступну форму:

```
[специфікатор-класса-памяти] [специфікатор-типа] имя-функции
([список-формальных-параметров])
{  тело-функции   }
```

Необов'язковий специфікатор-класса-памяти задає клас пам'яті функції, який може бути static або extern. Детально класи пам'яті будуть розглянуті в наступному розділі.

Специфікатор-типа функції задає тип повертаного значення і може задавати будь-який тип. Якщо специфікатор-типа не заданий, то передбачається, що функція повертає значення типу int.

Функція не може повертати масив або функцію, але може повертати покажчик на будь-який тип, у тому числі і на масив і на функцію. Тип повертаного значення, що задається у визначенні функції, повинен відповідати типу в оголошенні цієї функції.

Функція повертає значення якщо її виконання закінчується оператором return, що містить деяке вираження. Вказане вираження обчислюється, перетвориться, якщо необхідно, до типу повертаного значення і повертається в точку виклику функції в якості результату. Якщо оператор return не містить вирази або виконання функції завершується після виконання останнього її оператора (без виконання оператора return), то повертане значення не визначене. Для функцій, що не використовують повертане значення, має бути використаний тип void, що вказує на відсутність повертаного значення. Якщо функція визначена як функція, що повертає деяке значення, а в операторові return при виході з неї відсутнє вираження, то поведінка зухвалої функції після передачі їй управління може бути непередбачуваним. Функция возвращает значение если ее выполнение заканчивается

оператором return, содержащим некоторое выражение. Указанное выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения и возвращается в точку вызова функции в качестве результата. Если оператор return не содержит выражения или выполнение функции завершается после выполнения последнего ее оператора (без выполнения оператора return), то возвращаемое значение не определено. Для функций, не использующих возвращаемое значение, должен быть использован тип void, указывающий на отсутствие возвращаемого значения. Если функция определена как функция, возвращающая некоторое значение, а в операторе return при выходе из нее отсутствует выражение, то поведение вызывающей функции после передачи ей управления может быть непредсказуемым.

Список-формальных-параметрів - це послідовність оголошень формальних параметрів, розділена комами. Формальні параметри - це змінні, використовувані усередині тіла функції і набуваючі значення при виклику функції шляхом копіювання в них значень відповідних фактичних параметрів. Список-формальных-параметрів може закінчуватися комі (,) або комі з багатокрапкою (,...), це означає, що число аргументів функції змінне. Проте передбачається, що функція має, принаймні, стільки обов'язкових аргументів, скільки формальних параметрів задані перед останньою комою в списку параметрів. Такій функції може бути передане більше число аргументів, але над додатковими аргументами не проводиться контроль типів. Список-формальных-параметров - это последовательность объявлений формальных параметров, разделенная запятыми. Формальные параметры - это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений соответствующих фактических параметров. Список-формальных-параметров может заканчиваться запятой (,) или запятой с многоточием (,...), это означает, что число аргументов функции переменно. Однако предполагается, что функция имеет, по крайней мере, столько обязательных аргументов, сколько формальных параметров задано перед последней запятой в списке параметров. Такой функции может быть передано большее число аргументов, но над дополнительными аргументами не проводится контроль типов.

Якщо функція не використовує параметрів, то наявність круглих дужок обов'язкова, а замість списку параметрів рекомендується вказати слово void.

Порядок і типи формальних параметрів мають бути однаковими у визначенні функції і в усіх її оголошеннях. Типи фактичних параметрів при виклику функції мають бути сумісні з типами відповідних формальних параметрів. Тип формального параметра може бути будь-яким основним типом, структурою, об'єднанням, перерахуванням, покажчиком або масивом. Якщо тип формального параметра не вказаний, то цьому параметру привласнюється тип int. Порядок и типы формальных параметров должны быть одинаковыми в определении функции и во всех ее объявлениях. Типы фактических параметров при вызове функции должны быть совместимы с типами соответствующих формальных параметров. Тип формального параметра может быть любым основным типом, структурой, объединением, перечислением, указателем или массивом. Если тип формального параметра не указан, то этому параметру присваивается тип int.

Для формального параметра можна задавати клас пам'яті register, при цьому для величин типу int специфікатор типу можна опустити.

Ідентифікатори формальних параметрів використовуються в тілі функції в якості посилань на передані значення. Ці ідентифікатори не можуть бути перевизначені у блоці, що утворює тіло функції, але можуть бути перевизначені у внутрішньому блоці усередині тіла функції.

При передачі параметрів у функцію, якщо необхідно, виконуються звичайні арифметичні перетворення для кожного формального параметра і кожного фактичного параметра незалежно. Після перетворення формальний параметр не може бути коротший чим int, тобто оголошення формального параметра з типом char рівносильно його оголошенню з типом int. А параметри, що є дійсними числами, мають тип double. При передаче параметров в функцию, если необходимо, выполняются обычные арифметические преобразования для каждого формального параметра и каждого фактического параметра независимо. После преобразования формальный параметр не может быть короче чем int, т.е. объявление формального параметра с типом char равносильно его объявлению с типом int. А параметры, представляющие собой действительные числа, имеют тип double.

Перетворений тип кожного формального параметра визначає, як інтерпретуються аргументи, що поміщаються при виклику функції в стек. Невідповідність типів фактичних аргументів і формальних параметрів може бути причиною невірної інтерпретації.

Тіло функції - це складений оператор, що містить оператори, що визначають дію функції.

Усі змінні, оголошені в тілі функції без вказівки класу пам'яті, мають клас пам'яті auto, тобто вони є локальними. При виклику функції локальним змінним відводиться пам'ять в стеку і робиться їх ініціалізація. Управління передається першому операторові тіла функції і починається виконання функції, яке триває до тих пір, поки не зустрінеться оператор return або останній оператор тіла функції. Управління при цьому повертається в точку, що йде за точкою виклику, а локальні змінні стають недоступними. При новому виклику функції для локальних змінних пам'ять розподіляється знову, і тому старі значення локальних змінних втрачаються. Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти auto, т.е. они являются локальными. При вызове функции локальным переменным отводится память в стеке и производится их инициализация. Управление передается первому оператору тела функции и начинается выполнение функции, которое продолжается до тех пор, пока не встретится оператор return или последний оператор тела функции. Управление при этом возвращается в точку, следующую за точкой вызова, а локальные переменные становятся недоступными. При новом вызове функции для локальных переменных память распределяется вновь, и поэтому старые значения локальных переменных теряются.

Параметри функції передаються за значенням і можуть розглядатися як локальні змінні, для яких виділяється пам'ять при виклику функції і робиться ініціалізація значеннями фактичних параметрів. При виході з функції значення цих змінних втрачаються. Оскільки передача параметрів відбувається за значенням, в тілі функції не можна змінити значення змінних в зухвалій функції, що є фактичними параметрами. Проте, якщо в якості параметра передати покажчик на деяку змінну, то використовуючи операцію разадресации можна змінити значення цієї змінної. Параметры функции передаются по значению и могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и производится инициализация значениями фактических параметров. При выходе из функции значения этих переменных теряются. Поскольку передача параметров происходит по значению, в теле функции нельзя изменить значения переменных в вызывающей функции, являющихся фактическими параметрами. Однако, если в качестве параметра передать указатель на некоторую переменную, то используя операцию разадресации можно изменить значение этой переменной.

Приклад:

```
/*    Неправильне використання параметрів   */
void change (int x, int y)
{    int k=x;
         x=y;
         y=k;
}
```

У цій функції значення змінних x і y, що є формальними параметрами, міняються місцями, але оскільки ці змінні існують тільки усередині функції change, значення фактичних параметрів, використовуваних при виклику функції, залишаться незмінними. Для того, щоб мінялися місцями значення фактичних аргументів можна використати функцію приведену в наступному прикладі. В данной функции значения переменных x и y, являющихся формальными параметрами, меняются местами, но поскольку эти переменные существуют только внутри функции change, значения фактических параметров, используемых при вызове функции, останутся неизменными. Для того чтобы менялись местами значения фактических аргументов можно использовать функцию приведенную в следующем примере.

Приклад:

```
/*     Правильне використання параметрів   */
 void change (int *x, int *y)
 {  int k=*x;
       *x=*y;
       *y=k;
 }
```

При виклику такої функції в якості фактичних параметрів мають бути використані не значення змінних, а їх адреси

change (&a,&b);

Якщо вимагається викликати функцію до її визначення в даному файлі, або визначення функції знаходиться в іншому початковому файлі, то виклик функції слід упереджати оголошенням цієї функції. Оголошення (прототип) функції має наступний формат:

[специфікатор-класса-памяти] [специфікатор-типа] имя-функции ([список-формальных-параметров])[,список-имен-функций];

На відміну від визначення функції, в прототипі за заголовком відразу ж йде крапка з комою, а тіло функції відсутнє. Якщо декілька різних функцій повертають значення однакового типу і мають однакові списки формальних параметрів, то ці функції можна оголосити в одному прототипі, вказавши ім'я однієї з функцій в якості імені-функції, а усі інші помістити в список-имен-функций, причому кожна функція повинна супроводжуватися списком формальних параметрів. Правила використання інших елементів формату такі ж, як при визначенні функції. Імена формальних параметрів при оголошенні функції можна не вказувати, а якщо вони вказані, то їх зона дії поширюється тільки до кінця оголошення. В отличие от определения функции, в прототипе за заголовком сразу же следует точка с запятой, а тело функции отсутствует. Если несколько разных функций возвращают значения одинакового типа и имеют одинаковые списки формальных параметров, то эти функции можно объявить в одном прототипе, указав имя одной из функций в качестве имени-функции, а все другие поместить в список-имен-функций, причем каждая функция должна

сопровождаться списком формальных параметров. Правила использования остальных элементов формата такие же, как при определении функции. Имена формальных параметров при объявлении функции можно не указывать, а если они указаны, то их область действия распространяется только до конца объявления.

Прототип - це явне оголошення функції, яке передує визначенню функції. Тип повертаного значення при оголошенні функції повинен відповідати типу повертаного значення у визначенні функції.

Якщо прототип функції не заданий, а зустрівся виклик функції, то будується неявний прототип з аналізу форми виклику функції. Тип повертаного значення створюваного прототипу int, а список типів і числа параметрів функції формується на підставі типів і числа фактичних параметрів використовуваних при цьому виклику.

Таким чином, прототип функції необхідно задавати в наступних випадках:

1. Функція повертає значення типу, відмінного від int.

2. Потрібно проинициализировать деякий покажчик на функцію до того, як ця функція буде визначена.

Наявність в прототипі повного списку типів аргументів параметрів дозволяє виконати перевірку відповідності типів фактичних параметрів при виклику функції типам формальних параметрів, і, якщо необхідно, виконати відповідні перетворення.

У прототипі можна вказати, що число параметрів функції змінне, або що функція не має параметрів.

Якщо прототип заданий з класом пам'яті static, то і визначення функції повинне мати клас пам'яті static. Якщо специфікатор класу пам'яті не вказаний, то мається на увазі клас пам'яті extern.

Виклик функції має наступний формат:

адресное-выражение ([список-выражений])

Оскільки синтаксично ім'я функції є адресою початку тіла функції, в якості звернення до функції може бути використано адресное-выражение (у тому числі і ім'я функції або разадресация покажчика на функцію), значення адреси функції, що має.

Список-выражений є списком фактичних параметрів, що передаються у функцію. Цей список може бути і порожнім, але наявність круглих дужок обов'язкова.

Фактичний параметр може бути величиной будь-якого основного типу, структурою, об'єднанням, перерахуванням або покажчиком на об'єкт будь-якого типу. Масив і функція не можуть бути використані в якості фактичних параметрів, але можна використати покажчики на ці об'єкти.

Виконання виклику функції відбувається таким чином:

1. Обчислюються вираження в списку виразів і піддаються звичайним арифметичним перетворенням. Потім, якщо відомий прототип функції, тип отриманого фактичного аргументу порівнюється з типом відповідного формального параметра. Якщо вони не співпада-

ють, то або робиться перетворення типів, або формується повідомлення про помилку. Число виразів в списку виразів повинне співпадати з числом формальних параметрів, якщо тільки функція не має змінного числа параметрів. У останньому випадку перевірці підлягають тільки обов'язкові параметри. Якщо в прототипі функції вказано, що їй не потрібно параметри, а при виклику вони вказані, формується повідомлення про помилку. Вычисляются выражения в списке выражений и подвергаются обычным арифметическим преобразованиям. Затем, если известен прототип функции, тип полученного фактического аргумента сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо формируется сообщение об ошибке. Число выражений в списке выражений должно совпадать с числом формальных параметров, если только функция не имеет переменного числа параметров. В последнем случае проверке подлежат только обязательные параметры. Если в прототипе функции указано, что ей не требуются параметры, а при вызове они указаны, формируется сообщение об ошибке.

2. Відбувається привласнення значень фактичних параметрів відповідним формальним параметрам.

3. Управління передається на перший оператор функції.

4. Виконання оператора return в тілі функції повертає управління і можливо, значення в зухвалу функцію. За відсутності оператора return управління повертається після виконання останнього оператора тіла функції, а повертане значення не визначене.

Адресне вираження, що стоїть перед дужками визначає адресу функції, що викликається. Це означає що функція може бути викликана через покажчик на функцію.

Приклад:

```
int (*fun) (int x, int *y);
```

Тут оголошена змінна fun як покажчик на функцію з двома параметрами: типу int і покажчиком на int. Сама функція повинна повертати значення типу int. Круглі дужки, що містять ім'я покажчика fun і ознака покажчика *, обов'язкові, інакше запис

```
int *fun (intx, int *y);
```

інтерпретуватиметься як оголошення функції fun що повертає покажчик на int.

Виклик функції можливий тільки після ініціалізації значення покажчика fun і має вигляд:

```
(*fun)(i,&j);
```

У цьому виразі для отримання адреси функції, на яку посилається покажчик fun використовується операція разадресации *.

Покажчик на функцію може бути переданий в якості параметра функції. При цьому разадресация відбувається під час виклику функції, на яку посилається покажчик на функцію. Присвоїти значення покажчику на функцію можна в операторові привласнення, спожившиім'я функції без списку параметрів.

Приклад:

```
double (*fun1) (int x, int y);
double fun2(int k, int l);
   fun1=fun2;          /* ініціалізація покажчика на функцію */
   (*fun1)(2,7);       /* звернення до функції                */
```

У розглянутому прикладі покажчик на функцію fun1 описаний як покажчик на функцію з двома параметрами, що повертає значення типу double, і також описана функція fun2. Інакше, тобто коли покажчику на функцію привласнюється функція описана інакше, ніж покажчик, станеться помилка.

Розглянемо приклад використання покажчика на функцію в якості параметра функції що обчислює похідну від функції cos(x).

Приклад:

```
double proiz(double x, double dx, double (*f) (double x));
double fun(double z);
int main()
{
  double x;               /*  точка обчислення похідної */
  double dx;              /*  приріст                   */
  double z;               /*  значення похідної         */
  scanf("%f,%f",&x,&dx);  /*  введення значень x і dx         */
  z=proiz(x, dx, fun);      /*  виклик функції                 */
  printf("%f", z);         /*  друк значення похідної  */
  return 0;
}
double proiz(double x, double dx, double (*f) (double z))
{         /* функція обчислює похідну  */
   double xk, xk1, pr;
   xk=fun(x);
   xk1=fun(x+dx);
   pr=(xk1/xk - 1e0)*xk/dx;
   return pr;
}
double fun( double z)
{     /* функція від якої обчислюється похідна  */
   return (cos(z));
}
```

Для обчислення похідної від якої-небудь іншої функції можна змінити тіло функції fun або використати при виклику функції proiz ім'я іншої функції. Зокрема, для обчислення похідної від функції cos(x) можна викликати функцію proiz у формі

z=proiz(x, dx, cos);

а для обчислення похідної від функції sin(x) у формі

z=proiz(x, dx, sin);

Будь-яка функція в програмі на мові СІ може бути викликана рекурсивно, тобто вона може викликати саму себе. Компілятор допускає будь-яке число рекурсивних викликів. При кожному виклику для формальних параметрів і змінних з класом пам'яті auto і register виділяється нова область пам'яті, так що їх значення з попередніх викликів не втрачаються, але в кожен момент часу доступні тільки значення поточного виклику.

Змінні, оголошені з класом пам'яті static, не вимагають виділення нової області пам'яті при кожному рекурсивному виклику функції і їх значення доступні впродовж усього часу виконання програми.

Класичний приклад рекурсії - це математичне визначення факторіалу n! :

```
n! = 1 при n=0;
     n*(n - 1)!  при    n>1 .
```

Функція, що обчислює факторіал, матиме наступний вид:

```
long fakt(int n)
{
  return ( (n==1) ? 1: n*fakt(n - 1) );
}
```

Хоча компілятор мови СІ не обмежує число рекурсивних викликів функцій, це число обмежується ресурсом пам'яті комп'ютера і при занадто великому числі рекурсивних викликів може статися переповнювання стека.

1.5.2. Виклик функції зі змінним числом параметрів

При виклику функції зі змінним числом параметрів у виклику цієї функції задається будь-яке необхідне число аргументів. У оголошенні і визначенні такої функції змінне число аргументів задається багатокрапкою у кінці списку формальних параметрів або списку типів аргументів.

Усі аргументи, задані у виклику функції, розміщуються в стеку. Кількість формальних параметрів, оголошених для функції, визначається числом аргументів, які беруться із стека і привласнюються формальним параметрам. Програміст відповідає за правильність вибору додаткових аргументів із стека і визначення числа аргументів, що знаходяться в стеку.

Прикладами функцій зі змінним числом параметрів є функції з бібліотеки функцій мови СІ, що здійснюють операції введення-виведення інформації (printf, scanf і тому подібне). Детально ці функції розглянуті в третій частині книги.

Програміст може розробляти свої функції зі змінним числом параметрів. Для забезпечення зручного способу доступу до аргументів функції зі змінним числом параметрів є три макровизначення (макроси) va_start, va_arg, va_end, що знаходяться в заголовному файлі stdarg.h. Ці макроси вказують на те, що функція, розроблена користувачем, має деяке число обов'язкових аргументів, за якими йде змінне число необов'язкових аргументів. Обов'язкові аргументи доступні через свої імена як при виклику звичайної функції. Для витягання необов'язкових аргументів використовуються макроси va_start, va_arg, va_end в наступному порядку. Программист может разрабатывать свои функции с переменным числом параметров. Для обеспечения удобного способа доступа к аргументам функции с переменным числом параметров имеются три макроопределения (макросы) va_start, va_arg, va_end, находящиеся в заголовочном файле stdarg.h. Эти макросы указывают на то, что функция, разработанная пользователем, имеет некоторое число обязательных аргументов, за которыми следует переменное число необязательных аргументов. Обязательные аргументы доступны через свои имена как при вызове обычной функции. Для извлечения необязательных аргументов используются макросы va_start, va_arg, va_end в следующем порядке.

Макрос va_start призначений для установки аргументу arg_ptr на початок списку необов'язкових параметрів і має вигляд функції з двома параметрами:

void va_start(arg_ptr, prav_param);

Параметр prav_param має бути останнім обов'язковим параметром функції, що викликається, а покажчик arg_prt має бути оголошений з визначенням в списку змінних типу va_list у виді:

va_list arg_ptr;

Макрос va_start має бути використаний до першого використання макросу va_arg.

Макрокоманда va_arg забезпечує доступ до поточного параметра функції, що викликається, і теж має вигляд функції з двома параметрами

type_arg va_arg(arg_ptr, type);

Ця макрокоманда витягає значення типу type за адресою, заданою покажчиком arg_ptr, збільшує значення покажчика arg_ptr на довжину використаного параметра (довжина type) і таким чином параметр arg_ptr вказуватиме на наступний параметр функції, що викликається. Макрокоманда va_arg використовується стільки разів, скільки необхідно для витягання усіх параметрів функції, що викликається.

Макрос va_end використовується після закінчення обробки усіх параметрів функції і встановлює покажчик списку необов'язкових параметрів на нуль (NULL).

Розглянемо застосування цих макросів для обробки параметрів функції такою, що обчислює середнє значення довільної послідовності цілих чисел. Оскільки функція має змінне число параметрів вважатимемо кінцем списку значення рівне - 1. Оскільки в списку має бути хоч би один елемент, у функції буде один обов'язковий параметр.

Приклад:

```
#include
int main()
{ int n;
  int sred_znach(int,...);
  n=sred_znach(2,3,4,-1);
                 /* виклик з чотирма параметрами */
  printf("n=%d", n);
  n=sred_znach(5,6,7,8,9,-1);
                /* виклик з шістьма параметрами   */
  printf("n=%d", n);
  return (0);
}

int sred_znach(int x,...);
{
   int i=0,  j=0,  sum=0;
   va_list uk_arg;
   va_start(uk_arg, x);  /*  установка покажчика uk_arg на  */
                        /*  перший необязятельный параметр */
   if (x!=-1) sum=x;    /*  перевірка на порожнечу списку     */
   else return (0);
   j++;
```

```
    while ( (i=va_arg(uk_arg, int))!=-1)
                                /* вибірка чергового   */
    {                           /* параметра і перевірка */
      sum+=i;                   /* на кінець списку      */
      j++;
    }
    va_end(uk_arg);         /* закриття списку параметрів   */
    return (sum/j);
}
```

1.5.3. Передача параметрів функції main

Функція main, з якою починається виконання Сі-програми, може бути визначена з параметрами, які передаються із зовнішнього оточення, наприклад, з командного рядка. У зовнішньому оточенні діють свої правила представлення даних, а точніше, усі дані представляються у вигляді рядків символів. Для передачі цих рядків у функцію main використовуються два параметри, перший параметр служить для передачі числа передаваних рядків, другої для передачі самих рядків. Загальноприйняті (але не обов'язкові) імена цих параметрів argc і argv. Параметр argc має тип int, його значення формується з аналізу командного рядка і дорівнює кількості слів в командному рядку, включаючи і ім'я програми (під словом розуміється будь-який текст пропуск, що не містить символу), що викликається. Параметр argv це масив покажчиків на рядки, кожна з яких містить одне слово з командного рядка. Якщо слово повинне містити символ пропуск, то при записі його в командний рядок воно має бути поміщене в лапки. Функция main, с которой начинается выполнение СИ-программы, может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию main используются два параметра, первый параметр служит для передачи числа передаваемых строк, второй для передачи самих строк. Общепринятые (но не обязательные) имена этих параметров argc и argv. Параметр argc имеет тип int, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой программы (под словом понимается любой текст не содержащий символа пробел). Параметр argv это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ пробел, то при записи его в командную строку оно должно быть заключено в кавычки.

Функція main може мати і третій параметр, який прийнято називати argp, і який служить для передачі у функцію main параметрів операційної системи (середовища) в якій виконується Сі-програма.

Заголовок функції main має вигляд:

int main (int argc, char *argv[], char *argp[])

Якщо, наприклад, командний рядок Сі-програми має вигляд:

A:\>cprog working 'C program' 1

те аргументи argc, argv, argp представляються в пам'яті як показано в схемі на рис.1.

```
argc  [ 4  ]
argv  [    ]--> [    ]-->   [A:\cprog.exe\0]
                [    ]-->   [working\0]
                [    ]-->   [C program\0]
```

```
                  [    ]-->   [1\0]
                  [NULL]
argp  [    ]--> [    ]-->   [path=A:\;C:\\0]
                  [    ]-->   [lib=D:\LIB\0]
                  [    ]-->   [include=D:\INCLUDE\0]
                  [    ]-->   [conspec=C:\COMMAND.COM\]
                  [NULL]
```

Рис.1. Схема розміщення параметрів командного рядка

Операційна система підтримує передачу значень для параметрів argc, argv, argp, а на користувачі лежить відповідальність за передачу і використання фактичних аргументів функції main.

Наступний приклад представляє програму друку фактичних аргументів, що передаються у функцію main з операційної системи і параметрів операційної системи.

```
Приклад:
int main ( int argc, char *argv[], char *argp[])
{  int i=0;
   printf ("\n Ім'я програми %s", argv[0]);
   for    (i=1; i>=argc; i++)
   printf ("\n аргумент %d рівний %s", argv[i]);
   printf ("\n   Параметри операційної системи :");
   while  (*argp)
     {  printf ("\n %s",*argp);
        argp++;
     }
   return (0);
}
```

Доступ до параметрів операційної системи можна також отримати за допомогою бібліотечної функції geteuv, її прототип має наступний вигляд:

char *geteuv (const char *varname);

Аргумент цієї функції задає ім'я параметра середовища, покажчик на значення якої видасть функція geteuv. Якщо вказаний параметр не визначений в середовищі в даний момент, то повертане значення NULL.

Використовуючи покажчик, отриманий функцією geteuv, можна тільки прочитати значення параметра операційної системи, але не можна його змінити. Для зміни значення параметра системи призначена функція puteuv.

Компілятор мови СІ будує Сі-програму таким чином, що спочатку роботи програми виконується деяка ініціалізація, що включає, крім усього іншого, обробку аргументів, що передаються функції main, і передачу їй значень параметрів середовища. Ці дії виконуються бібліотечними функціями _setargv і _seteuv, які завжди поміщаються компілятором перед функцією main.

Якщо Сі-програма не використовує передачу аргументів і значень параметрів операційної системи, то доцільно заборонити використання бібліотечних функцій _setargv і _seteuv помістивши в Сі-програму перед функцією main функції з такими ж іменами, але що не виконують ніяких дій (заглушки). Початок програми в цьому випадку матиме вигляд:

```
_setargv()
 {  return ;    /* порожня функція */
```

```
 }
-seteuv()
 {  return ;    /* порожня функція */
 }
int main()
 {      /* головна функція без аргументів */
 ...
 ...
 renurn (0);
 }
```

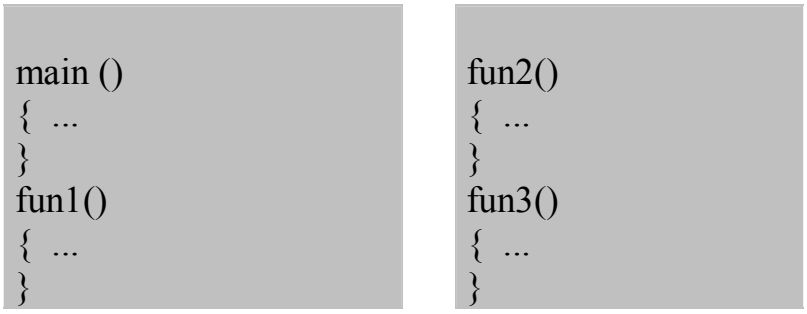
У приведеній програмі при виклику бібліотечних функцій _setargv і _seteuv будуть використані функції поміщені в програму користувачем і що не виконують ніяких дій. Це помітно понизить розмір отримуваного exe -файла.

1.6 СТРУКТУРА ПРОГРАМИ

1.6.1. Початкові файли і оголошення змінних

Звичайна Сі-програма є визначенням функції main, яка для виконання необхідних дій викликає інші функції. Наведені вище приклади програм були одним початковим файлом, що містить усі необхідні для виконання програми функції. Зв'язок між функціями здійснювався за даними за допомогою передачі параметрів і повернення значень функцій. Але компілятор мови СІ дозволяє також розбити програму на декілька окремих частин (початкових файлів), відтранслювати кожну частину окремо, і потім об'єднати усі частини в один виконуваний файл за допомогою редактора зв'язків. Обычная СИ-программа представляет собой определение функции main, которая для выполнения необходимых действий вызывает другие функции. Приведенные выше примеры программ представляли собой один исходный файл, содержащий все необходимые для выполнения программы функции. Связь между функциями осуществлялась по данным посредством передачи параметров и возврата значений функций. Но компилятор языка СИ позволяет также разбить программу на несколько отдельных частей (исходных файлов), оттранслировать каждую часть отдельно, и затем объединить все части в один выполняемый файл при помощи редактора связей.

При такій структурі початкової програми функції, що знаходяться в різних початкових файлах можуть використати глобальні зовнішні змінні. Усі функції в мові Сі за визначенням зовнішні і завжди доступні з будь-яких файлів. Наприклад, якщо програма складається з двох початкових файлів, як показано на Малюнок - 2., те функція main може викликати будь-яку з трьох функцій fun1, fun2, fun3, а кожна з цих функцій може викликати будь-яку іншу.



file1.c **file2.c**

Малюнок - 2. Приклад програми з двох файлів

Для того, щоб визначувана функція могла виконувати які або дії, вона повинна використати змінні. У мові СІ усі змінні мають бути оголошені до їх використання. Оголошення встановлюють відповідність імені і атрибутів змінної, функції або типу. Визначення змінної викликає виділення пам'яті для зберігання її значення. Клас пам'яті, що виділяється, визначається специфікатором класу пам'яті, і визначає час життя і зону видимості змінної, пов'язані з поняттям блоку програми. Для того, чтобы определяемая функция могла выполнять какие либо действия, она должна использовать переменные. В языке СИ все переменные должны быть объявлены до их использования. Объявления устанавливают соответствие имени и атрибутов переменной, функции или типа. Определение переменной вызывает выделение памяти для хранения ее значения. Класс выделяемой памяти определяется спецификатором класса памяти, и определяет время жизни и область видимости переменной, связанные с понятием блока программы.

У мові СІ блоком вважається послідовність оголошень, визначень і операторів, поміщена у фігурні дужки. Існують два види блоків - складений оператор і визначення функції, що складається із складеного оператора, що є тілом функції, і передування тілу заголовка функції (у який входять ім'я функції, типи повертаного значення і формальних параметрів). Блоки можуть включати складені оператори, але не визначення функцій. Внутрішній блок називається вкладеним, а зовнішній блок - охоплюючим. В языке СИ блоком считается последовательность объявлений, определений и операторов, заключенная в фигурные скобки. Существуют два вида блоков - составной оператор и определение функции, состоящее из составного оператора, являющегося телом функции, и предшествующего телу заголовка функции (в который входят имя функции, типы возвращаемого значения и формальных параметров). Блоки могут включать в себя составные операторы, но не определения функций. Внутренний блок называется вложенным, а внешний блок - объемлющим.

Час життя - це інтервал часу виконання програми, впродовж якого програмний об'єкт (змінна або функція) існує. Час життя змінної може бути локальним або глобальним. Змінна з глобальним часом життя має розподілену для неї пам'ять і певне значення протягом всього часу виконання програми, починаючи з моменту виконання оголошення цієї змінної. Змінна з локальним часом життя має розподілену для нього пам'ять і певне значення тільки під час виконання блоку, в якому ця змінна визначена або оголошена. При кожному вході у блок для локальної змінної розподіляється нова пам'ять, яка звільняється при виході з блоку. Время жизни - это интервал времени выполнения программы, в течение которого программный объект (переменная или функция) существует. Время жизни переменной может быть локальным или глобальным. Переменная с глобальным временем жизни имеет распределенную для нее память и определенное значение на протяжении всего времени выполнения программы, начиная с момента выполнения объявления этой переменной. Переменная с локальным временем жизни имеет распределенную для него память и определенное значение только во время выполнения блока, в котором эта переменная определена или объявлена. При каждом входе в блок для

локальной переменной распределяется новая память, которая освобождается при выходе из блока.

Усі функції в СІ мають глобальний час життя і існують впродовж усього часу виконання програми.

Зона видимості - це частина тексту програми, в якій може бути використаний цей об'єкт. Об'єкт вважається видимим у блоці або в початковому файлі, якщо в цьому блоці або файлі відомі ім'я і тип об'єкту. Об'єкт може бути видимим в межах блоку, початкового файлу або в усіх початкових файлах, що утворюють програму. Це залежить від того, на якому рівні оголошений об'єкт: на внутрішньому, тобто усередині деякого блоку, або на зовнішньому, тобто поза усіма блоками.

Якщо об'єкт оголошений усередині блоку, то він видимий в цьому блоці, і в усіх внутрішніх блоках. Якщо об'єкт оголошений на зовнішньому рівні, то він видимий від точки його оголошення до кінця цього початкового файлу.

Об'єкт може бути зроблений глобально видимим за допомогою відповідних оголошень в усіх початкових файлах, що утворюють програму.

Специфікатор класу пам'яті в оголошенні змінної може бути auto, register, static або extern. Якщо клас пам'яті не вказаний, то він визначається за умовчанням з контексту оголошення.

Об'єкти класів auto і register мають локальний час життя. Специфікатори static і extern визначають об'єкти з глобальним часом життя.

При оголошенні змінної на внутрішньому рівні може бути використаний будь-який з чотирьох специфікаторів класу пам'яті, а якщо він не вказаний, то мається на увазі клас пам'яті auto.

Змінна з класом пам'яті auto має локальний час життя і видна тільки у блоці, в якому оголошена. Пам'ять для такої змінної виділяється при вході у блок і звільняється при виході з блоку. При повторному вході у блок цієї змінної може бути виділена інша ділянка пам'яті.

Змінна з класом пам'яті auto автоматично не ініціалізувалася. Вона має бути проинициализирована явно при оголошенні шляхом привласнення їй початкового значення. Значення неініціалізованої змінної на клас пам'яті auto зважає невизначеним.

Специфікатор класу пам'яті register пропонує компілятору розподілити пам'ять для змінної в регістрі, якщо це представляється можливим. Використання регістрової пам'яті зазвичай призводить до скорочення часу доступу до змінної. Змінна, оголошена з класом пам'яті register, має ту ж зону видимості, що і змінна auto. Число регістрів, які можна використати для значень змінних, обмежене можливостями комп'ютера, і у тому випадку, якщо компілятор не має у розпорядженні вільних регістрів, то змінній виділяється пам'ять як для кла-

су auto. Клас пам'яті register може бути вказаний тільки для змінних з типом int або покажчиків з розміром, рівним розміру int. Спецификатор класса памяти register предписывает компилятору распределить память для переменной в регистре, если это представляется возможным. Использование регистровой памяти обычно приводит к сокращению времени доступа к переменной. Переменная, объявленная с классом памяти register, имеет ту же область видимости, что и переменная auto. Число регистров, которые можно использовать для значений переменных, ограничено возможностями компьютера, и в том случае, если компилятор не имеет в распоряжении свободных регистров, то переменной выделяется память как для класса auto. Класс памяти register может быть указан только для переменных с типом int или указателей с размером, равным размеру int.

Змінні, оголошені на внутрішньому рівні із специфікатором класу пам'яті static, забезпечую можливість зберегти значення змінної при виході з блоку і використати його при повторному вході у блок. Така змінна має глобальний час життя і зону видимості усередині блоку, в якому вона оголошена. На відміну від змінних з класом auto, пам'ять для яких виділяється в стеку, для змінних з класом static пам'ять виділяється в сегменті даних, і тому їх значення зберігається при виході з блоку. Переменные, объявленные на внутреннем уровне со спецификатором класса памяти static, обеспечиваю возможность сохранить значение переменной при выходе из блока и использовать его при повторном входе в блок. Такая переменная имеет глобальное время жизни и область видимости внутри блока, в котором она объявлена. В отличие от переменных с классом auto, память для которых выделяется в стеке, для переменных с классом static память выделяется в сегменте данных, и поэтому их значение сохраняется при выходе из блока.

Приклад:

```
/*  оголошення змінної і на внутрішньому рівні
        з  класом пам'яті static.                  */
/*   початковий файл file1.c       */
  main()
     { ...
     }
  fun1()
     {  static int i=0; ...
     }
 /* початковий файл   file2.c       */
  fun2()
     {  static int i=0; ...
     }
  fun3()
     {  static int i=0; ...
     }
```

У наведеному прикладі оголошені три різні змінні з класом пам'яті static, що мають однакові імена і. Кожна з цих змінних має глобальний час життя, але видима тільки в тому блоці (функції), в якій вона оголошена. Ці змінні можна використати для підрахунку числа звернень до кожної з трьох функцій.

Змінні класу пам'яті static можуть ініціалізувати константним вираженням. Якщо явної ініціалізації немає, то такій змінній привласнюється нульове значення. При ініціалізації константним адресним вираженням можна використати адреси будь-яких зовнішніх об'єктів, окрім адрес об'єктів з класом пам'яті auto, оскільки адреса останніх не є константою і змінюється при кожному вході у блок. Ініціалізація виконується один раз при першому вході у блок.

Змінна, оголошена локально з класом пам'яті extern, є посиланням на змінну з тим же самим ім'ям, визначену глобально в одному з початкових файлів програми. Мета такого оголошення полягає в тому, щоб зробити визначення змінної глобального рівня видимим усередині блоку.

Приклад:

```
/*  оголошення змінної i, що є ім'ям зовнішнього
    масиву довгих цілих чисел, на локальному рівні     */
/*     початковий файл    file1.c          */
  main()
    { ...
    }
  fun1()
    { extern long i[]; ...
    }
/*    початковий файл      file2.c         */
  long i[MAX]={0};
  fun2()
    {  ...
    }
  fun3()
    {  ...
    }
```

Оголошення змінної i[] як extern в наведеному прикладі робить її видимій усередині функції fun1. Визначення цієї змінної знаходиться у файлі file2.c на глобальному рівні і повинно бути тільки одне, тоді як оголошень з класом пам'яті extern може бути декілька.

Оголошення з класом пам'яті extern вимагається при необхідності використати змінну, описану в поточному початковому файлі, але нижче за текстом

програми, тобто до виконання її глобального визначення. Наступний приклад ілюструє таке використання змінної з ім'ям st.

Приклад:

```
main()
{ extern int st[]; ...
}
static int st[MAX]={0};
fun1()
{   ...
}
```

Оголошення змінної із специфікатором extern інформує компілятор про те, що пам'ять для змінної виділяти не вимагається, оскільки це виконано десь у іншому місці програми.

При оголошенні змінних на глобальному рівні може бути використаний специфікатор класу пам'яті static або extern, а так само можна оголошувати змінні без вказівки класу пам'яті. Класи пам'яті auto і register для глобального оголошення недопустимі.

Оголошення змінних на глобальному рівні - це або визначення змінних, або посилання на визначення, зроблені у іншому місці програми. Оголошення глобальної змінної, яке ініціалізувало цю змінну (явно або неявно), є визначенням змінної. Визначення на глобальному рівні може задаватися в наступних формах:

1. Змінна оголошена з класом пам'яті static. Така змінна може ініціалізувати явно константним вираженням, або за умовчанням нульовим значенням. Тобто обявления static int i=0 і static int і еквівалентні, і в обох випадках змінної і буде присвоєно значення 0.

2. Змінна оголошена без вказівки класу пам'яті, але з явною ініціалізацією. Такій змінній за умовчанням привласнюється клас пам'яті static. Тобто оголошення int i=1 і static int i=1 будуть еквівалентні.

Змінна оголошена глобально видима в межах залишку початкового файлу, в якому вона визначена. Вище за свій опис і в інших початкових файлах ця змінна невидима (якщо тільки вона не оголошена з класом extern).

Глобальна змінна може бути визначена тільки один раз в межах своєї зони видимості. У іншому початковому файлі може бути оголошена інша глобальна змінна з таким же ім'ям і з класом пам'яті static, конфлікту при цьому не виникає, оскільки кожна з цих змінних буде видимою тільки у своєму початковому файлі.

Специфікатор класу пам'яті extern для глобальних змінних використовується, як і для локального оголошення, в якості посилання на змінну, оголошену у іншому місці програми, тобто для розширення зони видимості змінної. При такому оголошенні зона видимості змінної розширюється до кінця початкового файлу, в якому зроблено оголошення.

У оголошеннях з класом пам'яті extern не допускається ініціалізація, оскільки ці оголошення посилаються на вже існуючі і визначені раніше змінні.

Змінна, на яку робиться посилання за допомогою специфікатора extern, може бути визначена тільки один раз в одному з початкових файлів програми.

1.6.2. Оголошення функцій

Функції завжди визначаються глобально. Вони можуть бути оголошені з класом пам'яті static або extern. Оголошення функцій на локальному і глобальному рівнях мають однаковий сенс.

Правила визначення зони видимості для функцій відрізняються від правил видимості для змінних і полягають в наступному.

1. Функція, оголошена як static, видима в межах того файлу, в якому вона визначена. Кожна функція може викликати іншу функцію з класом пам'яті static зі свого початкового файлу, але не може викликати функцію визначену з класом static в іншому початковому файлі. Різні функції з класом пам'яті static однакові імена, що мають, можуть бути визначені в різних початкових файлах, і це не веде до конфлікту.

2. Функція, оголошена з класом пам'яті extern, видима в межах усіх початкових файлів програми. Будь-яка функція може викликати функції з класом пам'яті extern.

3. Якщо в оголошенні функції відсутній специфікатор класу пам'яті, то за умовчанням приймається клас extern.

Усі об'єкти з класом пам'яті extern компілятор поміщає в об'єктному файлі в спеціальну таблицю зовнішніх посилань, яка використовується редактором зв'язків для дозволу зовнішніх посилань. Частина зовнішніх посилань породжується компілятором при зверненнях до бібліотечних функцій СІ, тому для дозволу цих посилань редакторові зв'язків мають бути доступні відповідні бібліотеки функцій.

1.6.3. Час життя і зона видимості програмних об'єктів

Час життя змінної (глобальною або локальною) визначається за наступними правилами.

1. Змінна, оголошена глобально (тобто поза усіма блоками), існує протягом всього часу виконання програми.

2. Локальні змінні (тобто оголошені усередині блоку) з класом пам'яті register або auto, мають час життя тільки на період виконання того блоку, в якому вони оголошені. Якщо локальна змінна оголошена з класом пам'яті static або extern, то вона має час життя на період виконання усієї програми.

Видимість змінних і функцій в програмі визначається наступними правилами.

1. Змінна, оголошена або визначена глобально, видима від точки оголошення або визначення до кінця початкового файлу. Можна зробити змінну видимою і в інших початкових файлах, для чого в цих файлах слід її оголосити з класом пам'яті extern.

2. Змінна, оголошена або визначена локально, видима від точки оголошення або визначення до кінця поточного блоку. Така змінна називається локальною.

3. Змінні з охоплюючих блоків, включаючи змінні оголошені на глобальному рівні, видимі у внутрішніх блоках. Цю видимість називають вкладеною. Якщо змінна, оголошена усередині блоку, має те ж ім'я, що і змінна, оголошена в охоплюючому блоці, то це різні змінні, і змінна з охоплюючого блоку у внутрішньому блоці буде невидимою.

4. Функції з класом пам'яті static видимі тільки в початковому файлі, в якому вони визначені. Всякі інші функції видимі в усій програмі.

Мітки у функціях видимі упродовж усієї функції.

Імена формальних параметрів, оголошені в списку параметрів прототипу функції, видимі тільки від точки оголошення параметра до кінця оголошення функції.

1.6.4. Ініціалізація глобальних і локальних змінних

При ініціалізації необхідно дотримуватися наступних правил:

1. Оголошення ті, що містять специфікатор класу пам'яті extern не можуть містити ініціаторів.

2. Глобальні змінні завжди ініціалізувалися, і якщо це не зроблено явно, то вони ініціалізувалися нульовим значенням.

3. Змінна з класом пам'яті static може ініціалізувати константним вираженням. Ініціалізація для них виконується один раз перед початком програми.

Якщо явна ініціалізація відсутня, то змінна ініціалізувалася нульовим значенням.

4. Ініціалізація змінних з класом пам'яті auto або register виконується всякий раз при вході у блок, в якому вони оголошені. Якщо ініціалізація змінних в оголошенні відсутня, то їх початкове значення не визначене.

5. Початковими значеннями для глобальних змінних і для змінних з класом пам'яті static мають бути константні вирази. Адреси таких змінних є константами і ці константи можна використати для ініціалізації оголошених глобально покажчиків. Адреси змінних з класом пам'яті auto або register не є константами і їх не можна використати в ініціаторах.

Приклад:

```
int global_var;
int func(void)
{ int local_var;                    /* за умовчанням auto  */
  static int *local_ptr=&local_var;   /*  так неправильно   */
  static int *global_ptr=&global_var; /*  а так правильно   */
  register int *reg_ptr=&local_var;   /*  і так правильно   */
}
```

У наведеному прикладі глобальна змінна global_var має глобальний час життя і постійну адресу в пам'яті, і цю адресу можна використати для ініціалізації статичного покажчика global_ptr. Локальна змінна local_var, що має клас пам'яті auto розміщується в пам'яті тільки на час роботи функції func, адреса цієї змінної не є константою і не може бути використаний для ініціалізації статичної змінної local_ptr. Для ініціалізації локальної регістрової змінної reg_ptr можна використати неконстантні вирази, і, зокрема, адресу змінної local_ptr. В приведенном примере глобальная переменная global_var имеет глобальное время жизни и постоянный адрес в памяти, и этот адрес можно использовать для инициализации статического указателя global_ptr. Локальная переменная local_var, имеющая класс памяти auto размещается в памяти только на время работы функции func, адрес этой переменной не является константой и не может быть использован для инициализации статической переменной local_ptr. Для инициализации локальной регистровой переменной reg_ptr можно использовать неконстантные выражения, и, в частности, адрес переменной local_ptr.

1.7.1. Методи доступу до елементів масивів

У мові СІ між покажчиками і масивами існує тісний зв'язок. Наприклад, коли оголошується масив у вигляді int array[25], те цим визначається не лише виділення пам'яті для двадцяти п'яти елементів масиву, але і для покажчика з ім'ям array, значення якого дорівнює адресі першого по рахунку (нульового) елементу масиву, тобто сам масив залишається безіменним, а доступ до елементів масиву здійснюється через покажчик з ім'ям array. З точки зору синтаксису мови покажчик array є константою, значення якої можна використати у виразах, але змінити це значення не можна. В языке СИ между указателями и

массивами существует тесная связь. Например, когда объявляется массив в виде int array[25], то этим определяется не только выделение памяти для двадцати пяти элементов массива, но и для указателя с именем array, значение которого равно адресу первого по счету (нулевого) элемента массива, т.е. сам массив остается безымянным, а доступ к элементам массива осуществляется через указатель с именем array. С точки зрения синтаксиса языка указатель arrey является константой, значение которой можно использовать в выражениях, но изменить это значение нельзя.

Оскільки ім'я масиву є покажчиком допустимо, наприклад, таке привласнення:

```
int arrey[25];
int *ptr;
ptr=array;
```

Тут покажчик ptr встановлюється на адресу першого елементу массива, причому привласнення ptr=arrey можна записати в еквівалентній формі ptr=&arrey[0].

Для доступу до елементів масиву існує два різні способи. Перший спосіб пов'язаний з використанням звичайних індексних виразів в квадратних дужках, наприклад, array[16]=3 або array[i+2]=7. При такому способі доступу записуються два вираження, причому друге вираження полягає в квадратні дужки. Один з цих виразів має бути покажчиком, а друге - вираженням цілого типу. Послідовність запису цих виразів може бути будь-якою, але в квадратних дужках записується вираження що наслідує другі. Тому записи array[16] і 16[array] будуть еквівалентними і означають елемент масиву з номером шістнадцять. Покажчик використовуваний в індексному вираженні не обов'язково має бути константою, що вказує на який-небудь масив, це може бути і змінна. Зокрема після виконання привласнення ptr=array доступ до шістнадцятого елементу масиву можна отримати за допомогою покажчика ptr у формі ptr[16] чи 16[ptr]. Для доступа к элементам массива существует два различных способа. Первый способ связан с использованием обычных индексных выражений в квадратных скобках, например, array[16]=3 или array[i+2]=7. При таком способе доступа записываются два выражения, причем второе выражение заключается в квадратные скобки. Одно из этих выражений должно быть указателем, а второе - выражением целого типа. Последовательность записи этих выражений может быть любой, но в квадратных скобках записывается выражение следующее вторым. Поэтому записи array[16] и 16[array] будут эквивалентными и обозначают элемент массива с номером шестнадцать. Указатель используемый в индексном выражении не обязательно должен быть константой, указывающей на какой-либо массив, это может быть и переменная. В частности после выполнения присваивания ptr=array доступ к шестнадцатому элементу массива можно получить с помощью указателя ptr в форме ptr[16] или 16[ptr].

Другий спосіб доступу до елементів масиву пов'язаний з використанням адресних виразів і операції разадресации у формі *(array+16)=3 або

*(array+i+2)=7. При такому способі доступу адресне вираження рівне адресі шістнадцятого елементу масиву теж може бути записано різними способами *(array+16) або *(16+array).

При реалізації на комп'ютері перший спосіб наводиться до другого, тобто індексне вираження перетвориться до адресного. Для наведених прикладів array[16] і 16[array] перетворяться в *(array+16).

Для доступу до початкового елементу масиву (тобто до елементу з нульовим індексом) можна використати просто значення покажчика array або ptr. Будь-яке з привласнень

```
*array = 2;
array[0] = 2;
*(array+0) = 2;
*ptr = 2;
ptr[0] = 2;
*(ptr+0) = 2;
```

привласнює початковому елементу масиву значення 2, але найшвидше виконаються привласнення *array=2 і *ptr=2, оскільки в них не вимагається виконувати операції складання.

1.7.2. Покажчики на багатовимірні масиви

Покажчики на багатовимірні масиви в мові СІ - це масиви масивів, тобто такі масиви, елементами яких є масиви. При оголошенні таких масивів в пам'яті комп'ютера створюється декілька різних об'єктів. Наприклад при виконанні оголошення двовимірного масиву int arr2[4][3] у пам'яті виділяється ділянка для зберігання значення змінної arr, яка є покажчиком на масив з чотирьох покажчиків. Для цього масиву з чотирьох покажчиків теж виділяється пам'ять. Кожен з цих чотирьох покажчиків містить адресу масиву з трьох елементів типу int, і, отже, в пам'яті комп'ютера виділяється чотири ділянки для зберігання чотирьох масивів чисел типу int, кожен з яких складається з трьох елементів. Таке виділення пам'яті показане на схемі на Малюнок - 3. Указатели на многомерные массивы в языке СИ - это массивы массивов, т.е. такие массивы, элементами которых являются массивы. При объявлении таких массивов в памяти компьютера создается несколько различных объектов. Например при выполнении объявления двумерного массива int arr2[4][3] в памяти выделяется участок для хранения значения переменной arr, которая является указателем на массив из четырех указателей. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа int, и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа int, каждый из которых состоит из трех элементов. Такое выделение памяти показано на схеме на Рисунок - 3.

arr

в

arr[0]	**a**	arr[0][0]	arr[0][1]	arr[0][2]
arr[1]	**a**	arr[1][0]	arr[1][1]	arr[1][2]
arr[2]	**a**	arr[2][0]	arr[2][1]	arr[2][2]
arr[3]	**a**	arr[3][0]	arr[3][1]	arr[3][2]

Малюнок - 3. Розподіл пам'яті для двовимірного масиву.

Таким чином, оголошення arr2[4][3] породжує в програмі три різні об'єкти: покажчик з ідентифікатором arr, безіменний масив з чотирьох покажчиків і безіменний масив з дванадцяти чисел типу int. Для доступу до безіменних масивів використовуються адресні вирази з покажчиком arr. Доступ до елементів масиву покажчиків здійснюється з вказівкою одного індексного вираження у формі arr2[2] чи *(arr2+2). Для доступу до елементів двовимірного масиву чисел типу int мають бути використані два індексні вираження у формі arr2[1][2] чи еквівалентних їй *(*(arr2+1)+2) і (*(arr2+1))[2]. Слід враховувати, що з точки зору синтаксису мови СІ покажчик arr і покажчики arr[0], arr[1], arr[2], arr[3] є константами і їх значення не можна змінювати під час виконання програми. Таким образом, объявление arr2[4][3] порождает в программе три разных объекта: указатель с идентификатором arr, безымянный массив из четырех указателей и безымянный массив из двенадцати чисел типа int. Для доступа к безымянным массивам используются адресные выражения с указателем arr. Доступ к элементам массива указателей осуществляется с указанием одного индексного выражения в форме arr2[2] или *(arr2+2). Для доступа к элементам двумерного массива чисел типа int должны быть использованы два индексных выражения в форме arr2[1][2] или эквивалентных ей *(*(arr2+1)+2) и (*(arr2+1))[2]. Следует учитывать, что с точки зрения синтаксиса языка СИ указатель arr и указатели arr[0], arr[1], arr[2], arr[3] являются константами и их значения нельзя изменять во время выполнения программы.

Розміщення тривимірного масиву відбувається аналогічно і оголошення float arr3[3][4][5] породжує в програмі окрім самого тривимірного масиву з шістдесяти чисел типу float масив з чотирьох покажчиків на тип float, масив з трьох покажчиків на масив покажчиків на float, і покажчик на масив масивів покажчиків на float.

При розміщенні елементів багатовимірних масивів вони розташовуються в пам'яті підряд по рядках, тобто найшвидше змінюється останній індекс, а повільніше - перший. Такий порядок дає можливість звертатися до будь-якого елементу багатовимірного масиву, використовуючи адресу його початкового елементу і тільки одне індексне вираження.

Наприклад, звернення до елементу arr2[1][2] можна здійснити за допомогою покажчика ptr2, оголошеного у формі int *ptr2=arr2[0] як звернення ptr2[1*4+2] (тут 1 і 2 це індекси використовуваного елементу, а 4 це число елементів в рядку) чи як ptr2[6]. Помітимо, що зовні схоже звернення arr2[6] виконати неможливо оскільки покажчика з індексом 6 не існує.

Для звернення до елементу arr3[2][3][4] з тривимірного масиву теж можно використати покажчик, описаний як float *ptr3=arr3[0][0] з одним індексним вираженням у формі ptr3[3*2+4*3+4] чи ptr3[22].

Далі приведена функція, що дозволяє знайти мінімальний елемент в тривимірному масиві. У функцію передається адреса початкового елементу і розміри масиву, повертане значення - покажчик на структуру, що містить індекси мінімального елементу.

```
struct INDEX {  int i
                int j
                int k }  min_index ;

struct INDEX  * find_min (int *ptr1, int l, int m int n)
{     int  min, i, j, k, ind;
      min=*ptr1;
      min_index.i=min_index.j=min_index.k=0;
      for (i=0; i*(ptr1+ind)
          {  min=*(ptr1+ind);
             min_index.i=i;
             min_index.j=j;
             min_index.k=k;
          }
      }
      return &min_index;
}
```

1.7.3. Операції з покажчиками

Над покажчиками можна виконувати унарні операції: інкремент і декремент. При виконанні операцій ++ і -- значення покажчика збільшується або зменшується на довжину типу, на який посилається використовуваний покажчик.

Приклад:

```
int *ptr, a[10];
  ptr=&a[5];
  ptr++;      /* дорівнює адресі елементу a[6] */
  ptr--;      /* дорівнює адресі елементу a[5] */
```

У бінарних операціях складання і віднімання можуть брати участь покажчик і величина типу int. При цьому результатом операції буде покажчик на початковий тип, а його значення буде на вказане число елементів більше або менше за початковий.

Приклад:

```
int *ptr1, *ptr2, a[10];
int i=2;
  ptr1=a+(i+4);   /*  дорівнює адресі елементу a[6] */
  ptr2=ptr1 - i;    /*  дорівнює адресі елементу a[4] */
```

У операції віднімання можуть брати участь два покажчики на один і той же тип. Результат такої операції має тип int і дорівнює числу елементів початкового типу між зменшуваним і таким, що віднімається, причому якщо перша адреса молодша, то результат має негативне значення.

Приклад:

```
int *ptr1, *ptr2, a[10];
int i;
  ptr1=a+4;
  ptr2=a+9;
  i=ptr1 - ptr2;  /*   рівне 5   */
  i=ptr2 - ptr1;  /*   рівно - 5  */
```

Значення двох покажчиків на однакові типи можна порівнювати в операціях ==, !=, <=, >, >= при цьому значення покажчиків розглядаються просто як цілі числа, а результат порівняння дорівнює 0 (брехня) або 1 (істина).

Приклад:

```
int *ptr1, *ptr2, a[10];
  ptr1=a+5;
  ptr2=a+7;
  if (prt1>ptr2) a[3]=4;
```

У цьому прикладі значення ptr1 менше значення ptr2 і тому оператор a[3]=4 не буде виконаний.

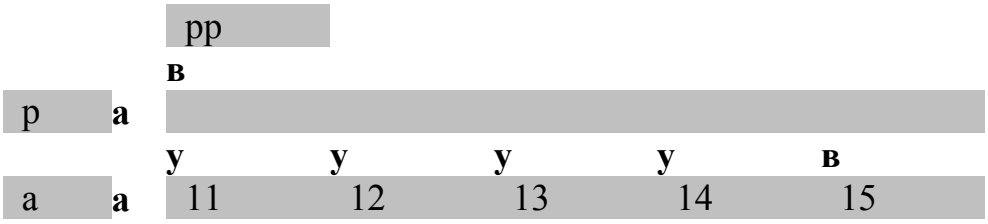
1.7.4. Масиви покажчиків

У мові СІ елементи масивів можуть мати будь-який тип, і, зокрема, можуть бути покажчиками на будь-який тип. Розглянемо декілька прикладів з використанням покажчиків.

Наступні оголошення змінних

```
int a[]={10,11,12,13,14,};
int *p[]={a, a+1, a+2, a+2, a+3, a+4};
int **pp=p;
```

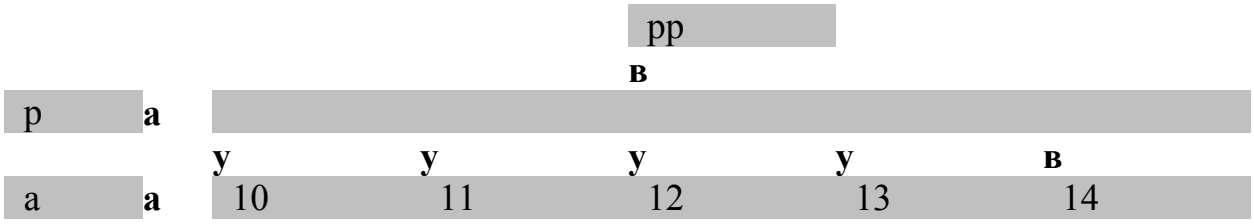
породжують програмні об'єкти, представлені на схемі на Малюнок, - 4.



Малюнок - 4. Схема розміщення змінних при оголошенні.

При виконанні операції pp - p отримаємо нульове значення, оскільки посилання pp і p рівні і вказують на початковий елемент масиву покажчиків, пов'язаного з покажчиком p (на елемент p[0]).

Після виконання операції pp+=2 схема зміниться і набере вигляду, зображеного на Малюнок, - 5.



Малюнок - 5. Схема розміщення змінних після виконання операції pp+=2.

Результатом виконання віднімання pp - p буде 2, оскільки значення pp є адреса третього елементу масиву p. Посилання *pp - a теж дає значення 2, оскільки звернення *pp є адреса третього елементу масиву a, а звернення a є адреса початкового елементу масиву a. При зверненні за допомогою посилання **pp отримаємо 12 - це значення третього елементу масиву a. Посилання *pp++ дасть значення четвертого елементу масиву p тобто адреса четвертого елементу масиву a.

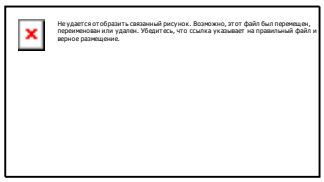
Якщо вважати, що pp=p, то звернення *++pp це значення першого елементу масиву a (тобто значення 11), операція ++*pp змінить вміст покажчика p[0], таким чином, що він стане рівним значенню адреси елементу a[1].

Складні звернення розкриваються зсередини. Наприклад звернення *(++(*pp)) можна розбити на наступні дії: *pp дає значення початкового елементу масиву p[0], далі це значення инкременируется ++(*p) внаслідок чого покажчик p[0] дорівнюватиме значенню адреси елементу a[1], і остання дія це вибірка значення за отриманою адресою, тобто значення 11.

У попередніх прикладах був використаний одновимірний масив, розглянемо тепер приклад з багатовимірним масивом і покажчиками. Наступні оголошення змінних

```
int a[3][3]={  { 11,12,13 },
               { 21,22,23 },
               { 31,32,33 }   };
int *pa[3]={ a, a[1],a[2] };
int *p=a[0];
```

породжують в програмі об'єкти представлені на схемі на Малюнок - 6.



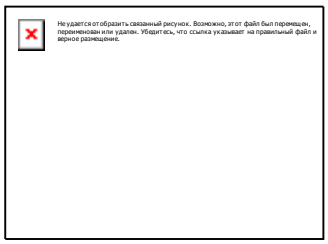
Малюнок - 6. Схема розміщення покажчиків на двовимірний масив.

Згідно з цією схемою доступ до елементу a[0][0] отримати по покажчиках a, p, pa за допомогою наступних посилань: a[0][0], *a, **a[0], *p, **pa, *p[0].

Розглянемо тепер приклад з використанням рядків символів. Оголошення змінних

```
char *c[]={ "abs", "dx", "yes", "no" };
char **cp[]={ c+3, c+2, c+1, c };
char ***cpp=cp;
```

можна зображувати схемою представленою на Малюнок - 7.



Малюнок - 7. Схема розміщення покажчиків на рядки.

1.7.5. Динамічне розміщення масивів

При динамічному розподілі пам'яті для масивів слід описати відповідний покажчик і привласнювати йому значення за допомогою функції calloc. Одновимірний масив a[10] з елементів типу float можна створити таким чином

```
float *a;
```

```
a=(float*)(calloc(10, sizeof(float));
```

Для створення двовимірного масиву спочатку треба розподілити пам'ять для масиву покажчиків на одновимірні масиви, а потім розподіляти пам'ять для одновимірних масивів. Нехай, наприклад, вимагається створити масив a[n][m], це можна зробити за допомогою наступного фрагмента програми :

```
#include
main ()
   {  double **a;
     int n, m, i;
     scanf("%d %d",&n,&m);
     a=(double **) calloc(m, sizeof(double *));
     for (i=0; i<=m; i++)
        a[i]=(double *)calloc(n, sizeof(double));
    . . . . . .
     }
```

Аналогічним чином можна розподілити пам'ять і для тривимірного масиву розміром n, m, l. Слід тільки пам'ятати, що непотрібну для подальшого виконання програми пам'ять слід звільняти за допомогою функції free.

```
#include
main ()
   {  long ***a;
     int n, m, l, i, j;
     scanf("%d %d %d",&n,&m,&l);
     /* --------  розподіл пам'яті -------- */
     a=(long ***) calloc(m, sizeof(long **));
     for (i=0; i<=m; i++)
        {    a[i]=(long **)calloc(n, sizeof(long *));
             for (j=0; i<=l; j++)
             a[i][j]=(long *)calloc(l, sizeof(long));
         }
    . . . . . .
     /* --------- звільнення пам'яті ----------*/
     for (i=0; i<=m; i++)
        {   for (j=0; j<=l; j++)
            free (a[i][j]);
            free (a[i]);
         }
      free (a);
     }
```

Розглянемо ще один цікавий приклад, в якому пам'ять для масивів розподіляється у функції, що викликається, а використовується в тій, що викликає. У такому разі у функцію, що викликається, вимагається передавати покажчики, яким будуть присвоєні адреси пам'яті, що виділяється для масивів.

Приклад:

```
#include
main()
{   int vvod(double ***, long **);
    double **a;     /*  покажчик для масиву  a[n][m]   */
    long *b;        /*  покажчик для масиву  b[n]      */
    vvod (&a,&b);
     ..   /* у функцію vvod передаються адреси покажчиків, */
     ..   /* а не їх значення                              */
     ..
}
int vvod(double ***a, long **b)
{   int n, m, i, j;
    scanf (" %d %d ",&n,&m);
    *a=(double **) calloc(n, sizeof(double *));
    *b=(long *) calloc(n, sizeof(long));
    for (i=0; i<=n; i++)
       *a[i]=(double *)calloc(m, sizeof(double));
    .....
 }
```

Відмітимо також ту обставину, що покажчик на масив не обов'язково повинен показувати на початковий елемент деякого масиву. Він може бути зрушений так, що початковий елемент матиме індекс відмінний від нуля, причому він може бути як позитивним так і негативним.

Приклад:

```
#include
int main()
{  float *q, **b;
   int i, j, k, n, m;
   scanf("%d %d",&n,&m);
   q=(float *) calloc(m, sizeof(float));
/* зараз покажчик q показує на початок масиву    */
   q[0]=22.3;
   q-=5;
/* тепер початковий елемент масиву має індекс 5,   */
/* а кінцевий елемент індекс n - 5                  */
```

```
  q[5]=1.5;
/* зрушення індексу не призводить до перерозподілу      */
/* масиву в пам'яті і зміниться початковий елемент     */
  q[6]=2.5;   /*  -  це другий елемент              */
  q[7]=3.5;   /*  -  це третій елемент              */
  q+=5;
/* тепер початковий елемент знову має індекс 0,     */
/* а значення елементів q[0], q[1], q[2] рівні       */
/* відповідно 1.5, 2.5, 3.5                          */
  q+=2;
/* тепер початковий елемент має індекс - 2,          */
/* наступний - 1, потім 0 і так далі по порядку      */
  q[- 2]=8.2;
  q[- 1]=4.5;
  q-=2;
/* повертаємо початкову індексацію, три перших       */
/* елементу масиву q[0], q[1], q[2], мають           */
/* значення 8.2, 4.5, 3.5                            */
  q--;
/* знову змінимо індексацію .                        */
/* Для звільнення області пам'яті в якій розміщений */
/* масив q використовується функція free(q), але оскільки */
/* значення покажчика q зміщене, те виконання     */
/* функції free(q) приведе до непередбачуваних наслідків. */
/* Для правильного виконання цієї функції    */
/* покажчик q має бути повернений в первинне */
/* положення                                         */
  free(++q);
/* Розглянемо можливість зміни індексації і      */
/* звільнення пам'яті для двовимірного масиву        */
  b=(float **) calloc(m, sizeof(float *));
  for (i=0; i < m; i++)
     b[i]=(float *)calloc(n, sizeof(float));
/* Після розподілу пам'яті початковим елементом      */
/* масиву буде елемент b[0][0]                       */
/* Виконаємо зрушення індексів так, щоб початковим       */
/* елементом став елемент b[1][1]                    */
  for (i=0; i < m ; i++) --b[i];
  b--;
/* Тепер присвоїмо кожному елементу масиву суму його  */
/* індексів                                          */
  for (i=1; i<=m; i++)
     for (j=1; j<=n; j++)
        b[i][j]=(float)(i+j);
/* Зверніть увагу на початкові значення лічильників   */
/* циклів і і j, він починаються з 1 а не з 0          */
```

```
  /* Повернемося до колишньої індексації              */
     for (i=1; i<=m; i++) ++b[i];
     b++;
  /* Виконаємо звільнення пам'яті                     */
     for (i=0; i < m; i++)  free(b[i]);
     free(b);
     ...
     ...
     return 0;
  }
```

В якості останнього прикладу розглянемо динамічний розподіл пам'яті для масиву покажчиків на функції, що мають один вхідний параметр типу double і повертають значення типу double.

Приклад:

```
#include
#include
 double cos(double);
 double sin(double);
 double tan(double);
int main()
{ double (*(*masfun)) (double);
  double x=0.5, y;
  int i;
  masfun=(double(*(*)) (double))
         calloc(3, sizeof(double(*(*)) (double)));
  masfun[0]=cos;
  masfun[1]=sin;
  masfun[2]=tan;
  for (i=0; i
```

1.8. Директиви Препроцесора

Директиви препроцесора є інструкціями, записаними в тексті програми на CI, і виконувані до трансляції програми. Директиви препроцесора дозволяють змінити текст програми, наприклад, замінити деякі лексеми в тексті, вставити текст з іншого файлу, заборонити трансляцію частини тексту і тому подібне. Усі директиви препроцесора розпочинаються зі знаку #. Після директив препроцесора крапка з комою не ставляться.

1.8.1. Директива #include

Директива #include включає в текст програми вміст вказаного файлу. Ця директива має дві форми:

```
#include "ім'я файлу"
#include
```

Ім'я файлу повинне відповідати угодам операційної системи і може складатися або тільки з імені файлу, або з імені файлу з передуванням йому маршрутом. Якщо ім'я файлу вказане в лапках, то пошук файлу здійснюється відповідно до заданого маршруту, а при його відсутності в поточному каталозі. Якщо ім'я файлу задане в кутових дужках, то пошук файлу здійснюється в стандартних директоріях операційної системи, що задаються командою PATH.

Директива #include може бути вкладеною, тобто у файлі, що включається, теж може міститися директива #include, яка заміщається після включення файлу, що містить цю директиву.

Директива #include широко використовується для включення в програму так званих заголовних файлів, що містять прототипи бібліотечних функцій, і тому більшість програм на CI розпочинаються з цієї директиви.

1.8.2. Директива #define

Директива #define служить для заміни констант, що часто використовуються, ключових слів, операторів або виразів деякими ідентифікаторами. Ідентифікатори, замінюючі текстові або числові константи, називають іменованими константами. Ідентифікатори, замінюючі фрагменти програм, називають макровизначеннями, причому макровизначення можуть мати аргументи.

Директива #define має дві синтаксичні форми:

```
#define  ідентифікатор текст
#define  ідентифікатор (список параметрів) текст
```

Ця директива замінює усі подальші входження ідентифікатора на текст. Такий процес називається макропідстановкою. Текст може бути будь-яким фрагментом програми на CI, а також може бути і відсутнім. У останньому випадку усі екземпляри ідентифікатора видаляються з програми.

Приклад:

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
```

Ці директиви змінять в тексті програми кожне слово WIDTH на число 80, а кожне слово LENGTH на вираження (80+10) разом з дужками, що оточують його.

Дужки, що містяться в макровизначенні, дозволяють уникнути непорозумінь, пов'язаних з порядком обчислення операцій. Наприклад, за відсутності дужок вираження t=LENGTH*7 буде перетворено у вираження t=80+10*7, а не у вираження t=(80+10)*7, як це виходить за наявності дужок, і в результаті вийде 780, а не 630.

У другій синтаксичній формі в директиві #define є список формальних параметрів, який може містити один або декілька ідентифікаторів, розділених комами. Формальні параметри в тексті макровизначення відмічають позиції на які мають бути підставлені фактичні аргументи макровиклику. Кожен формальний параметр може з'явитися в тексті макровизначення кілька разів. Во второй синтаксической форме в директиве #define имеется список формальных параметров, который может содержать один или несколько идентификаторов, разделенных запятыми. Формальные параметры в тексте макроопределения отмечают позиции на которые должны быть подставлены фактические аргументы макровызова. Каждый формальный параметр может появиться в тексте макроопределения несколько раз.

При макровиклику услід за ідентифікатором записується список фактичних аргументів, кількість яких повинна співпадати з кількістю формальних параметрів.

Приклад:

```
#define  MAX(x, y)  ((x)>(y))?(x):(y)
```

Ця директива замінить фрагмент

```
        t=MAX(i, s[i]);
```

на фрагмент

```
        t=((i)>(s[i])?(i) :(s[i]);
```

Як і в попередньому прикладі, круглі дужки, в які поміщені формальні параметри макровизначення, дозволяють уникнути помилок пов'язаних з неправильним порядком виконання операцій, якщо фактичні аргументи є виразами.

Наприклад, за наявності дужок фрагмент

```
        t=MAX(i&j, s[i]||j);
```

буде замінений на фрагмент

```
        t=((i&j)>(s[i]||j)?(i&j) :(s[i]||j);
```

а за відсутності дужок - на фрагмент

```
t=(i&j>s[i]||j)?i&j: s[i]||j;
```

у якому умовне вираження обчислюється в абсолютно іншому порядку.

1.8.3. Директива #undef

Директива #undef використовується для відміни дії директиви #define. Синтаксис цієї директиви наступний #undef ідентифікатор

Директива відміняє дію поточного визначення #define для вказаного ідентифікатора. Не є помилкою використання директиви #undef для ідентифікатора, який не був визначений директивою #define.

Приклад:

```
#undef WIDTH
#undef MAX
```

Ці директиви відміняють визначення іменованої константи WIDTH і макровизначення MAX.

2. Організація списків і їх обробка

2.1. Лінійні списки

2.1.1. Методи організації і зберігання лінійних списків

Лінійний список - це кінцева послідовність однотипних елементів (вузлів), можливо, з повтореннями. Кількість елементів в послідовності називається завдовжки списку, причому довжина в процесі роботи програми може змінюватися.

Лінійний список F, що складається з елементів D1, D2,...,Dn, записують у вигляді послідовності значень ув'язненого в кутові дужки F=, або представляють графічно (см.Рисунок - 12).

D1 **а** D2 **а** D3 **а** ... **а** Dn

Малюнок - 12. Зображення лінійного списку.

Наприклад, F1=< 2,3,1>,F2=< 7,7,7,2,1,12 >, F3=< >. Довжина списків F1, F2, F3 рівна відповідно до 3,6,0.

При роботі зі списками на практиці найчастіше доводиться виконувати наступні операції:

- знайти елемент із заданою властивістю;

- визначити перший елемент в лінійному списку;

- вставити додатковий елемент або після вказаного вузла;

- виключити певний елемент зі списку;

- упорядкувати вузли лінійного списку в певному порядку.

У реальних мовах програмування немає якої-небудь структури даних для представлення лінійного списку так, щоб усі вказані операції над ним виконувалися однаковою мірою ефективно. Тому при роботі з лінійними списками важливим є представлення використовуваних в програмі лінійних списків так, щоб була забезпечена максимальна ефективність і за часом виконання програми, і за об'ємом необхідної пам'яті. В реальных языках программирования нет какой-либо структуры данных для представления линейного списка так, чтобы все указанные операции над ним выполнялись в одинаковой степени эффективно. Поэтому при работе с линейными списками важным является представление используемых в программе линейных списков таким образом, чтобы была обеспечена максимальная эффективность и по времени выполнения программы, и по объему требуемой памяти.

Методи зберігання лінійних списків розділяються на методи послідовного і пов'язаного зберігання. Розглянемо прості варіанти цих методів для списку з цілими значеннями F=.

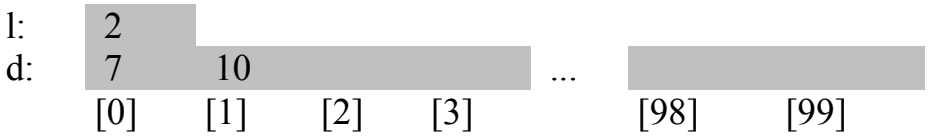
При послідовному зберіганні елементи лінійного списку розміщуються в масиві d фіксованих розмірів, наприклад, 100, і довжина списку вказується в змінній l, тобто в програмі необхідно мати оголошення виду

```
float d[100];   int l;
```

Розмір масиву 100 обмежує максимальні розміри лінійного списку. Список F в масиві d формується так:

```
d[0]=7; d[1]=10; l=2;
```

Отриманий список зберігається в пам'яті згідно з схемою на Малюнок - 13.



Малюнок - 13. Послідовне зберігання лінійного списку.

При пов'язаному зберіганні як елементи зберігання використовуються структури, пов'язані по одній з компонент в ланцюжок, на початок якої (першу структуру) вказує покажчик dl. Структура утворює елемент зберігання, повинна окрім відповідного елементу списку містити і покажчик на сусідній елемент зберігання.

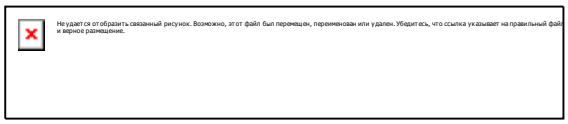
Опис структури і покажчика в цьому випадку може имееть вид:

```
typedef struct snd   /* структура елементу зберігання   */
{ float  val;       /* елемент списку                  */
 struct snd *n ;     /* покажчик на елемент зберігання */
} DL;
DL *p;              /* покажчик поточного елементу   */
DL *dl;             /* покажчик на початок списку    */
```

Для виділення пам'яті під елементи зберігання необхідно користуватися функцією malloc(sizeof(DL)) або calloc(l, sizeof(DL)). Формування списку в пов'язаному зберіганні може здійснюється операторами:

```
p=malloc(sizeof(DL));
p ->val=10;   p ->n=NULL;
dl=malloc(sizeof(DL));
dl ->val=7;   dl ->n=p;
```

У останньому елементі зберігання (кінець списку) покажчик на сусідній елемент має значення NULL. Отримуваний список зображений на Малюнок - 14.



Малюнок - 14. Зв'язне зберігання лінійного списку.

2.1.2. Операції зі списками при послідовному зберіганні

При виборі методу зберігання лінійного списку слід враховувати, які операції виконуватимуться і з якою частотою, час їх виконання і об'єм пам'яті, потрібний для зберігання списку.

Нехай є лінійний список з цілими значеннями і для його зберігання використовується масив d (з числом елементів 100), а кількість елементів в списку вказується змінній l. Реалізація вказаних раніше операцій над списком представляється наступними фрагментами програм які використовують оголошення:

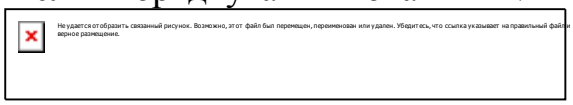
```
    float d[100];
    int i, j, l;
1) друк значення першого елементу (вузла)
  if (il) printf("\n немає елементу");
  else printf("d[%d]=%f ", i, d[i]);
2) видалення елементу, що йде за і -тым  вузлом
```

```
if (i>=l) printf("\n немає наступного ");
l--;
for (j=i+1;j<=1 || i>=l) printf("\n немає сусіда");
else printf("\n %d  %d", d[i - 1],d[i+1]);
```

4) додавання нового елементу new за i -тым вузлом

```
if (i==l || i>l) printf("\n не можна додати");
else
{    for (j=l; j>i+1; j--) d[j+1]=d[j];
     d[i+1]=new; l++;
}
```

5) часткове впорядкування списку з елементами K1, K2,...,Kl в список K1 ', K2...,Ks, K1, Kt",...,Kt", s+t+1=l так, щоб K1'=K1; після впорядкування покажчик v вказує на елемент K1' (див. Малюнок - 19)



Малюнок - 19. Схема часткового впорядкування списку.

```
ND *v;
float k1;
k1=dl ->val;
r=dl;
while( r ->n!=NULL )
{ v=r ->n;
  if (v ->valn=v ->n;
      v ->n=dl;
      dl=v;
    }
   else r=v;
}
```

Кількість дій, потрібних для виконання вказаних операцій над списком

у пов'язаному зберіганні, оцінюється співвідношеннями: для операцій 1 і 2 - Q=l; для

операцій 3 і 4 - Q=1; для операції 5 - Q=l.

2.1.4. Організація двозв'язкових списків

Пов'язане зберігання лінійного списку називається списком з двома зв'язками або

двозв'язковим списком, якщо кожен елемент зберігання має два компоненти покажчика

(посилання на попередній і подальший елементи лінійного списку).

У програмі двозв'язковий список можна реалізувати за допомогою описів:

```
typedef struct ndd
  { float val;      /* значення елементу              */
    struct ndd * n; /* покажчик на наступний елемент   */
    struct ndd * m; /* покажчик на предыдующий елемент */
    } NDD;
NDD * dl, * p, * r;
```

Графічна інтерпретація методу пов'язаного зберігання списку F=< 2,5,7,1 >

як списку з двома зв'язками приведена на Малюнок - 20.

Малюнок - 20. Схема зберігання двозв'язкового списку.

Вставка нового вузла зі значенням new за елементом, визначуваним покажчиком p

здійснюється за допомогою операторів:

```
r=malloc(NDD);
r ->val=new;
r ->n=p ->n;
(p ->n) ->m=r;
p ->=r;
```

Видалення елементу, що йде за вузлом, на який вказує p

```
p ->n=r;
p ->n=(p ->n) ->n;
```

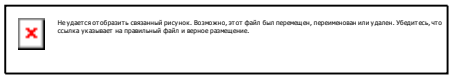
```
( (p ->n) ->n ) ->m=p;
free(r);
```

Пов'язане зберігання лінійного списку називається циклічним списком, якщо

його останній вказує на перший елемент, а покажчик dl - на останній

елемент списку.

Схема циклічного зберігання списку F=< 2,5,7,1 > приведена на Малюнок - 21.



Малюнок - 21. Схема циклічного зберігання списку.

При рішенні конкретних завдань можуть виникати різні види пов'язаного зберігання.

Нехай на вході задана послідовність цілих чисел B1, B2,...,Bn з

інтервалу від 1 до 9999, і нехай Fi (1<i за збільшенням. Скласти

процедуру для формування Fn в пов'язаному зберіганні і повернення покажчика на

його початок.

При рішенні задачі в кожен момент часу маємо впорядкований список Fi і

при введенні елементу Bi+1 вставляємо його в потрібне місце списку Fi, отримуючи

впорядкований список Fi+1. Тут можливі три варіанти: в списку немає елементів;

число вставляється в початок списку; число вставляється в кінець списку. Щоб

уніфікувати усі можливі варіанти, початковий список організовуємо як пов'язаний

список з двох елементів .

Розглянемо програму рішення поставленої задачі, в якій покажчики dl

r, p, v мають наступне значення: dl вказує початок списку; p, v - два сусідніх вузла; r фіксує вузол, що містить чергове введене значення in.

```
#include
#include
typedef struct str1
  { float val;
    struct str1 *n; }  ND;
main()
 {  ND *arrange(void);
    ND *p;
    p=arrange();
    while(p!=NULL)
      {
      printf("\n %f ", p ->val);
      p=p ->n;
      }
 }
ND *arrange() /*   формування впорядкованого списку */
 {  ND *dl, *r, *p, *v;
    float in=1;
    char *is;
    dl=malloc(sizeof(ND));
    dl ->val=0;                /*  перший елемент     */
    dl ->n=r=malloc(sizeof(ND));
    r ->val=10000; r ->n=NULL;    /*  останній елемент  */
    while(1)
    {
       scanf(" %s", is);
       if(* is=='q') break;
       in=atof(is);
       r=malloc(sizeof(ND));
       r ->val=in;
       p=dl;
       v=p ->n;
       while(v ->valn;
       }
       r ->n=v;
       p ->n=r;
   }
   return(dl);
 }
```

2.1.5. Стеки і черги

Залежно від методу доступу до елементів лінійного списку розрізняють різновиди лінійних списків називаються стеком, чергою і двосторонньою чергою.

Стік - це кінцева послідовність деяких однотипних елементів - скалярних змінних, масивів, структур або об'єднань, серед яких можуть бути і однакові. Стек позначається у виді: S= і представляє динамічну структуру даних; її кількість елементів заздалегідь не вказується і в процесі роботи, як правило змінюється. Якщо в стеку елементів немає, то він називається порожнім і позначається S=< >.

Допустимими операціями над стеком є:

- перевірка стека на порожнечу S=< >,
- додавання нового елементу Sn+1 в кінець стека - перетворення < S1,...,Sn> у < S1,...,Sn+1>;
- вилучення останнього елементу із стека - перетворення < S1,...,Sn - 1, Sn> у < S1,...,Sn - 1>;
- доступ до його останнього елементу Sn, якщо стек не порожній.

Таким чином, операції додавання і видалення елементу, а також доступу до елементу виконуються тільки у кінці списку. Стек можна представити як стопку книг на столі, де додавання або узяття нової книги можливе тільки згори.

Черга - це лінійний список, де елементи видаляються з початку списку, а додаються у кінці списку (як звичайна черга в магазині).

Двостороння черга - це лінійний список, у якого операції додавання

і видалення елементів і доступу до елементів можливі як спочатку так і у кінці

списку. Таку чергу можна представити як послідовність книг тих, що стоять на

полиці, так що доступ до них можливий з обох кінців.

Реалізація стеків і черг в програмі може бути виконана у виді

послідовного або пов'язаного зберігання. Розглянемо приклади організації

стека цими способами.

Однією з форм представлення виразів є польський інверсний запис

задаюча вираження так, що операції в нім записуються в порядку виконання

а операнди знаходяться безпосередньо перед операцією.

Наприклад, вираження

(6+8)*5-6/2

у польському інверсному записі має вигляд

6 8 + 5 * 6 2 / -

Особливість такого запису полягає в тому, що значення вираження можна

вичислити за один перегляд записи зліва направо, використовуючи стек, який до

цього має бути порожній. Кожне нове число заноситься в стек, а операції

виконуються над верхніми елементами стека, замінюючи ці елементи результатом

операції. Для приведеного вираження динаміка зміни стека матиме вигляд

S =< >; ; ; ; ; ;
; ; ; .

Нижче приведена функція eval, яка обчислює значення вираження, заданого у масиві m у формі польського інверсного запису, причому m[i]>0 означає ненегативне число, а значення m[i]

```
float eval (float *m, int l)
{ int p, n, i;
  float stack[50],c;
   for(i=0; i < l ;i++)
   if ((n=m[i])
```

Розглянемо інше завдання. Нехай вимагається ввести деяку послідовність символів, що закінчується точкою, і надрукувати її в зворотному порядку (тобто якщо на вході буде "ABcEr-1". те на виході має бути "1-rEcBA"). Представлена нижче програма спочатку вводить усі символи послідовності, записуючи їх в стік, а потім вміст стека друкується в зворотному порядку. Це основна особливість стека - чим пізніше елемент занесений в стек, тим раніше він буде витягнутий із стека. Реалізація стека виконана в пов'язаному зберіганні при допомозі покажчиків p і q на тип, що іменується ім'ям STACK.

```
#include
typedef struct st          /* оголошення типу STACK */
  { char ch;
    struct st *ps;   } STACK;
main()
{   STACK *p,*q;
    char a;
    p=NULL;
    do                  /*  заповнення стека    */
    { a=getch();
      q=malloc(sizeof(STR1));
       q ->ps=p; p=q;
       q ->ch=a;
```

```
    }  while(a!='.');
    do                  /*  друк стека        */
    { p=q ->ps;free(q);q=p;
      printf("%c", p ->ch);
    } while(p ->ps!=NULL);
 }
```

2.1.6. Стисле і індексне зберігання лінійних списків

При зберіганні великих об'ємів інформації у формі лінійних списків небажано зберігати елементи з однаковим значенням, тому використовують різні методи стискування списків.

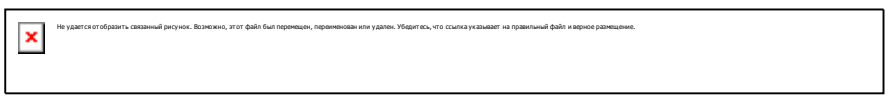
Стисле зберігання. Нехай в списку B= декілька елементів мають однакове значення V, а список B'= виходить з B заміною кожного елементу Ki на пару Ki'=(i, Ki). Нехай далі B"=< K1", K2",...,Km" > - підсписок B ', що виходить викреслюванням усіх пар Ki=(i, V). Стислим зберіганням B являється метод зберігання B", в якому елементи зі значенням V умовчуються.

Розрізняють послідовне стисле зберігання і пов'язане стисле зберігання.

Наприклад, для списку B=, що містить декілька вузлів з значенням X, послідовне стисле і пов'язане стисле зберігання, із замовчуванням елементів зі значенням X, представлені на Малюнок - 22,23.

1,C	3,Y	6,S	7,H	9,T

Малюнок - 22. Послідовне стисле зберігання списку.



Малюнок - 23. Зв'язне стисле зберігання списку.

Гідність стислого зберігання списку при великому числі елементів зі значенням

V полягає в можливості зменшення об'єму пам'яті для його зберігання.

Пошук і -го елементу в пов'язаному стислому зберіганні здійснюється методом

повного перегляду, при послідовному зберіганні - методом бінарного пошуку.

Переваги і недоліки послідовного стислого і пов'язаного стислого

зберігань аналогічні перевагам і недолікам послідовного і пов'язаного

зберігань.

Розглянемо наступне завдання. На вході задані дві послідовності цілих

чисел M=< M1, M2,...,M10000 >, N=< N1, N2,...,N10000 >, причому 92% елементів

послідовності M дорівнюють нулю. Скласти програму для обчислення суми

творів Mi * Ni, i=1,2,...,10000.

Припустимо, що список M зберігається послідовно стисло в масиві структур

m з оголошенням:

```
struct
{ int nm;
  float val; }   m[10000];
```

Для визначення кінця списку додамо ще один елемент з порядковим номером

m[j].nm=10001, який називається стоппером (stopper) і розташовується за

останнім елементом стислого зберігання списку в масиві m.

Програма для знаходження шуканої суми має вигляд:

```
# include
main()
{ int i, j=0;
  float inp, sum=0;
  struct                    /* оголошення  масиву  */
  { int nm;                   /* структур            */
    float val;  }    m[10000];

  for(i=0;i
```

Індексне зберігання використовується для зменшення часу пошуку потрібного

елементу в списку і полягає в наступному. Початковий список B =

< K1, K2, ...,Kn > розбивається на декілька підсписків B1, B2, ...,Bm таким

образом, що кожен елемент списку B потрапляє тільки в один з підсписків, і

додатково використовується індексний список з M елементами, що вказують на

початок списків B1, B2, ...,Bm.

Вважається, що список зберігається індексний за допомогою підсписків B1, B2, ...,Bm

і індексного списка X =< ADG1, ADG2,... ADGm >, де ADGj - адреса початку

підсписку Bj, j=1, M.

При індексному зберіганні елемент До підсписку Bj має індекс j. Для отримання

індексного зберігання початковий список B часто перетвориться в список B' шляхом

включення в кожен вузол ще і його порядкового номера в початковому списку B, а в

j -й елемент індексного списку X, окрім ADGj, може включатися деяка

додаткова інформація про підсписок Bj. Розбиття списку B на підсписки

здійснюється так, щоб усі елементи B, що мають певну властивість Pj

потрапляли в один підсписок Bj.

Гідністю індексного зберігання є те, що для знаходження елементу До

із заданою властивістю Pj досить проглянути тільки елементи підсписку Bj; його

початок знаходиться за індексним списком X, оскільки для будь-кого До, що належить

Bi, при і не рівному j властивість Pj не виконується.

У розбитті B часто використовується індексна функція G(K), що обчислює по

елементу До його індекс j, тобто G(K)=j. Функція G зазвичай залежить від позиції До

що означає поз.K, в підсписку B або від значення певної частини компоненти

До - її ключа.

Розглянемо список B=< K1, K2, ...,K9 > з елементами

K1=(17, Y), K2=(23, H), K3=(60, I), K4=(90, S), K5=(66, T)
K6=(77, T), K7=(50, U), K8=(88, W), K9=(30, S).

Якщо для розбиття цього списку на підсписки в якості індексної функції

узяти Ga(K)=1+(поз.K - 1)/3, то список розділиться на три підсписки:

B1a=
B2a=
B3a=.

Додаючи усюди ще і початкову позицію елементу в списку, отримуємо:

B1a'=
B2a'=
B3a'=.

Якщо в якості індексної функції вибрати іншу функцію Gb(K)=1+(поз.K - 1)%3

те отримаємо списки:

B1b"=
B2b"=
B3b" =.

Тепер для знаходження вузла K6 досить проглянути тільки одну з трьох

послідовностей (списків). При використанні функції Ga(K) це список B2a

а при функції Gb(K) список B3b".

Для індексної функції Gc(K)=1+K1/100, де K1 - перша компонента елементу До

знаходимо:

B1=
B2=
B3=
B4=.

Щоб знайти тут вузол з першим компонентом-ключем K1=77, досить

проглянути список B2.

При реалізації індексного зберігання застосовується методика A для зберігання

індексного списку X (функція Ga(X)) і методика C для зберігання підсписків

B1, B2,...,Bm (функція Gc(Bi)), тобто використовується, так зване, A - C індексне

зберігання.

У практиці часто використовується послідовно-пов'язане індексне зберігання.

Оскільки зазвичай довжина списку індексів відома, то його зручно зберігати

послідовно, забезпечуючи прямий доступ до будь-якого елементу списку індексів.

Підсписки B1, B2,...,Bm зберігаються пов'язаний, що спрощує вставку і видалення

вузлів(елементів). Зокрема, подібний метод зберігання використовується в ЄС ЕОМ

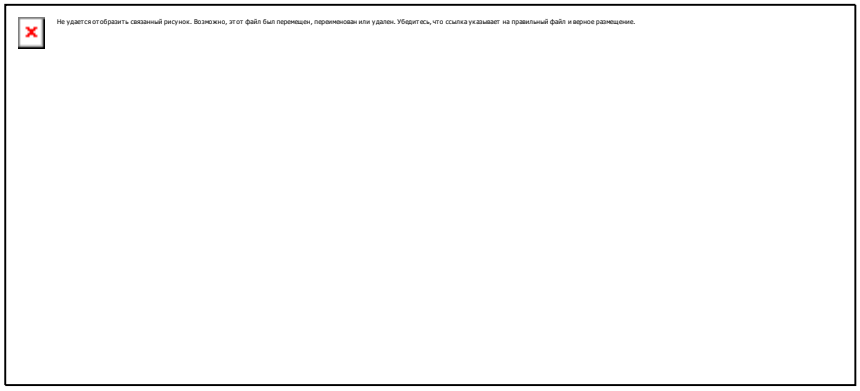
для організації, так званих, індексно-послідовних наборів даних, в

яких доступ до окремих записів можливий як послідовно, так і при

допомозі ключа.

Послідовно-пов'язане індексне зберігання для наведеного прикладу

зображено на Малюнок - 24, де X=.



Малюнок - 24. Послідовно-пов'язане індексне зберігання списку.

Розглянемо ще одне завдання. На вході задана послідовність цілих

позитивних чисел, що закінчується нулем. Скласти процедуру для введення цій

послідовності і організації її послідовно-пов'язаного індексного

зберігання так, щоб числа, співпадаючі в двох останніх цифрах

поміщалися в один підсписок.

Виберемо в якості індексної функції G(K)=K%100+1, а в якості індексного

списку X - масив з 100 елементів. Наступна функція вирішує поставлену

завдання:

```
#include
#include
typedef struct nd
        {  float val;
           struct nd *n; }    ND;
int index (ND *x[100])
{  ND *p;
   int i, j=0;
   float inp;
   for (i=0; ival=inp;
      p ->n=x[i];
      x[i]=p;
      scanf("%d",&inp);
   }
   return j;
}
```

Повертаним значенням функції index буде число оброблених елементів

списку.

Для індексного списку також може використовуватися індексне зберігання. Нехай

наприклад, є список B= з елементами

K1=(338, Z), K2=(145, A), K3=(136, H), K4=(214, I), K5 =(146, C)
K6=(334, Y), K7=(333, P), K8=(127, G), K9=(310, O), K10=(322, X).

Вимагається розділити його на сім підсписків, тобто X= таким

образом, щоб в кожен список B1, B2,...,B7 потрапляли елементи, співпадаючі в

першій компоненті першими двома цифрами. Список X, у свою чергу, буде-мо

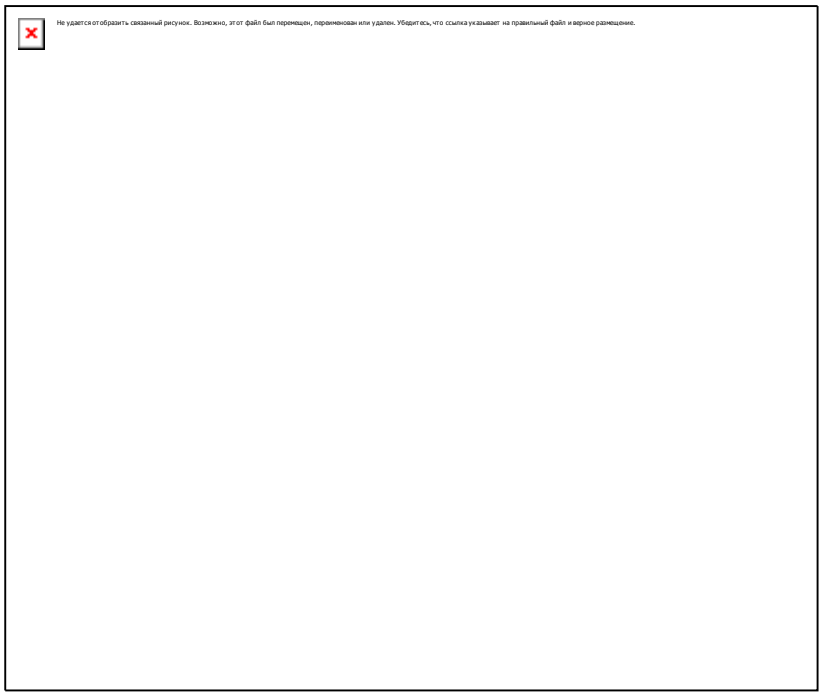
індексувати списком індексів Y=, щоб в кожен список Y1, Y2, Y3

потрапляли елементи з X, у яких в першій компоненті співпадають перші цифри.

Якщо списки B1, B2,...,B7 зберігати пов'язаний, а списки індексів X, Y індексний, то

такий спосіб зберігання списку B називається зв'язано-зв'язаним пов'язаним

індексним зберіганням. Графічне зображення цього зберігання приведене на Малюнок - 25.



Малюнок - 25. зв'язано-зв'язане пов'язане індексне зберігання списку.

2.2. Сортування І Злиття Списків

При роботі зі списками дуже часто виникає необхідність перестановки елементів списку в певному порядку. Таке завдання називається сортуванням списку і для її вирішення існують різні методи. Розглянемо деякі з них.

2.2.1. Бульбашкове сортування

Завдання сортування полягає в наступному: заданий список цілих чисел (простий випадок) B=< K1, K2,..., Kn >. Вимагається переставити елементи

списку так, щоб отримати впорядкований список B'=< K'1, K'2,...,K'n >, у якому для будь-кого 1<=i<=n елемент K'(i)<= K'(i+1).

При обмінному сортуванні впорядкований список B' виходить з B систематичним обміном пари елементів, що поруч стоять, не відповідають необхідному порядку, поки такі пари існують.

Найбільш простий метод систематичного обміну сусідніх елементів з неправильним порядком при перегляді усього списку ліворуч на право визначає бульбашкове сортування: максимальні елементи як би спливають у кінці списку.

Приклад:

B=, початковий список;
B1=<- 5,10,8,7,20>, перший перегляд;
B2=<- 5,8,7,10,20>, другий перегляд;
B3=<- 5,7,8,10,20>, третій перегляд.

У подальших прикладах вважатимемо, що сортується одновимірний масив (або його частина від індексу n до індексу m) в порядку зростання елементів.

Нижчеприведена функція bubble сортує вхідний масив методом бульбашкового сортування.

```
/* сортування бульбашковим методом        */
float * bubble(float * a, int m, int n)
{
char is=1;
int i;
float c;
  while(is)
  {   is=0;
      for (i=m+1; i<=n; i++)
      if ( a[i] < a[i - 1] )
      {    c=a[i];
           a[i]=a[i - 1];
           a[i - 1]=c;
           is=1;
      }
  }
  return(a);
}
```

Бульбашкове сортування виконується при кількості дій Q=(n - m)* (n - m) і не вимагає додаткової пам'яті.

2.2.2. Сортування вставкою

Впорядкований масив B' виходить з B таким чином: спочатку він складається з єдиного елементу K1; далі для i=2,...,N виконується вставка вузла Ki в B' так, що B' залишається впорядкованим списком довжини i.

Наприклад, для початкового списку B=< 20,-5,10,8,7 > маємо:

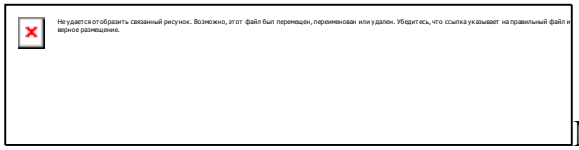
B=< 20,-5,10,8,7> B'=< >
B=< - 5,10,8,7 > B'=< 20 >
B=< 10,8,7 > B'=< - 5,20 >
B=< 8,7 > B'=< - 5,10,20 >
B=< 7 > B'=< - 5,8,10,20 >
B=< > B'=< - 5,7,8,10,20 >

Функція insert реалізує сортування вставкою.

```
/*  сортування методом вставки          */
float *insert(float *s, int m, int n)
{
int i, j, k;
float aux;
for (i=m+1; i<=n; i++)
    { aux=s[i];
      for (k=m; k<=i && s[k]=k; j--)  s[j+1]=s[j];
      s[k]=aux;
    }
    return(a);
}
```

Тут обидва списки B і B' розміщуються в масиві s, причому список B займає частину s с індексами від i до n, а B' - частина s с індексами від m до i - 1 (див. Малюнок - 26).

При сортуванні вставкою потрібно Q=(n - m)* (n - m) порівнянь і не потрібно додаткову пам'ять.



Малюнок - 26. Схема руху індексів при сортуванні вставкою.

2.2.3. Сортування за допомогою вибору

Впорядкований список B' виходить з У багатократним застосуванням вибірки з B мінімального елементу, видаленням цього елементу з B і додаванням його в кінець списку B', який спочатку має бути порожнім.

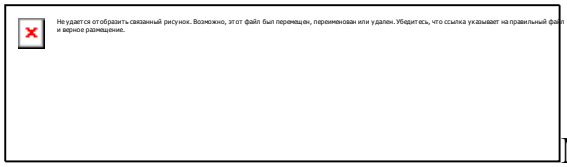
Наприклад:

B=, B'=< >
B=, B'=<- 5>
B=, B'=<- 5,7>
B=, B'=<- 5,7,8>
B=, B'=<- 5,7,8,10>
B=< >, B'=<- 5,7,8,10,20> .

Функція select упорядковує масив s сортуванням за допомогою вибору.

```
/*  сортування методом вибору        */
double *select( double *s, int m, int n)
{
 int i, j;
 double c;
 for (i=m; i<=n;j++)
    if(s[i]>s[j])
     {  c=s[i];
       s[i]=s[j];
       s[j]=c;
     }
 return(s);
}
```

Тут, як і в попередньому прикладі обидва списки B і B' розміщуються в різних частинах масиву s (див. Малюнок - 27). При сортуванні за допомогою вибору потрібно Q=(n - m)* (n - m) дій і не потрібно додаткову пам'ять.



Малюнок - 27. Схема руху індексів при сортуванні вибором.

Сортування квадратичною вибіркою. Початковий список B з N елементів ділиться на M підсписків B1, B2,...,Bm, де M рівно квадратному кореню з N, і в кожному B1 знаходиться мінімальний елемент G1. Найменший елемент усього списку У визначається як мінімальний елемент Gj в списку, і вибраний елемент Gj замінюється новим найменшим зі списку Bj. Кількість дій, потрібна для сортування квадратичною вибіркою, дещо менше, ніж в

попередніх методах Q= N*N, але потрібно додаткову пам'ять для зберігання списку G. Сортировка квадратичной выборкой. Исходный список B из N элементов делится на M подсписков B1,B2,...,Bm, где M равно квадратному корню из N, и в каждом B1 находится минимальный элемент G1. Наименьший элемент всего списка B определяется как минимальный элемент Gj в списке , и выбранный элемент Gj заменяется новым наименьшим из списка Bj. Количество действий, требуемое для сортировки квадратичной выборкой, несколько меньше, чем в предыдущих методах Q= N*N, но требуется дополнительная память для хранения списка G.

2.2.4. Злиття списків

Впорядковані списки A і B довжин M і N зливаються в один впорядкований список З довжини M+N, якщо кожен елемент з A і У входить в З точно один раз. Так, злиття списків A= і B= з 5 і 4 елементів дає в якості результату список З= з 9 елементів.

Для злиття списків A і B список Із спочатку покладається порожнім, а потім до нього послідовно приписується перший вузол з A або B, що виявився меншим і відсутній в C.

Складемо функцію для злиття двох впорядкованих, розташованих рядом частин масиву s. Параметром цієї функції буде початковий масив s з виділеними в нім двома розташованими поруч впорядкованими підмасивами : перший з індексу low до індексу low+l, другий з індексу low+l+1 до індексу up, де змінні low, l, up вказують місцерозташування підмасивів. Функція merge здійснює злиття цих підмасивів, утворюючи на їхньому місці впорядкований масив з індексами від low до up (див. Малюнок - 28).

```
/*   слияние списков */
double *merge(double *s, int low, int up, int l)
{
double *b,*c,v;
  int i,j,k;
  b=calloc(l,sizeof(double));
  c=calloc(up+1-l,sizeof(double));
  for(i=low;is[up-1]) ?
                    (s[low+l-1]+1) : (s[up-1]+1)));
  i=(j=0);
  k=low;
  while(b[i]<v||c[j]
```

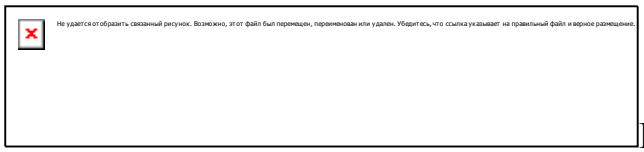


Рисунок - 28. Схема движения индексов при слиянии списков.

2.2.5. Сортировка списков путем слияния

Для получения упорядоченного списка B' последовательность значений B= разделяют на N списков B1=, B2=,...,Bn=, длина каждого из которых 1. Затем осуществляется функция прохода, при которой M>=2 упорядоченных списков B1,B2,...,Bm заменяется на M/2 (или (M+1)/2) упорядоченных списков, B(2i-1)-ого и B(2i)-ого (2i<=M) и добавлением Bm при нечетном M. Проход повторяется до тех пор пока не получится одна последовательность длины N.

Приведем пример сортировки списка путем использования слияния, отделяя последовательности косой чертой, а элементы запятой.

Пример:

```
 9 / 7 / 18 / 3 / 52 / 4 / 6 / 8 / 5 / 13 / 42 / 30 / 35 / 26;
7,9 / 3,18 / 4 / 52 / 6 / 8 / 54 / 13 / 30 / 42 / 26 / 35;
3,7,9,18 /  4,6,8,52 /  5,13,30,42 /  26,35;
3,4,6,7,8,9,18,52 /   5,13,26,30,35,42;
3,4,5,6,7,8,9,13,18,26,30,35,42,52.
```

Количество действий, требуемое для сортировки слиянием, равно Q=N*log2(N), так как за один проход выполняется N сравнений, а всего необходимо осуществить log2(N) проходов. Сортировка слиянием является очень эффективной и часто применяется для больших N, даже при использовании внешней памяти.

Функция smerge упорядочивает массив s сортировкой слиянием, используя описанную ранее функцию merge.

```
/* сортировка слиянием */
double *smerge (double *s, int m, int n)
{ int l,low,up;
  double *merge (double *, int, int, int);
  l=1;
  while(l<=(n-m))
  {  low=m;
     up=m-1;
     while (l+up < n)
     {   up=(low+2*l-1 < n) ? (low+2*l-1) : n ;
```

```
        merge (s,low,up,l);
        low=up-1;
      }
      l*=2;
    }
   return(a);
  }
```

Для сортировки массива путем слияния удобно использовать рекурсию. Составим рекурсивную функцию srecmg для сортировки массива либо его части. При каждом вызове сортируемый массив делится на две равных части, каждая из которых сортируется отдельно, а затем происходит их слияние, как это показано на Рисунок - 29.

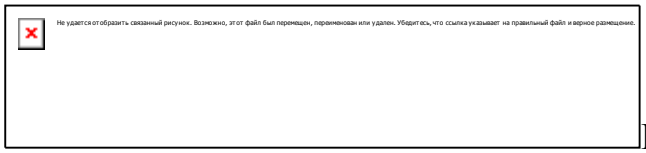


Рисунок - 29. Схема сортировки слиянием.

```
/*   рекурсивная   сортировка слиянием  1/2          */
double *srecmg (double *a, int m, int n)
{ double * merge (double *, int, int, int);
  double * smerge (double *, int, int);
   int i;
  if (n>m)
     { i=(n+m)/2;
       srecmg(a,m,i);
       srecmg(a,i+1,n);
       merge(a,m,n,(n-m)/2+1);
     }
  return (a);
}
```

2.2.6. Быстрая и распределяющая сортировки

Быстрая сортировка состоит в том, что список B=< K1,K2,...,Kn> реорганизуется в список B',< K1 >,B", где B' - подсписок B с элементами, не большими K1, а B" - подсписок B с элементами, большими K1. В списке B',< K1 >,B" элемент K1 расположен на месте, на котором он должен быть в результирующем отсортированном списке. Далее к спискам B' и B" снова применяется упорядочивание быстрой сортировкой. Приведем в качестве примера сортировку списка, отделяя упорядоченные элементы косой чертой, а элементы Ki знаками < и >.

Пример:

9, 7, 18, 3, 52, 4, 6, 8, 5, 13, 42, 30, 35, 26
7, 3, 4, 6, 8, 5/ / 18, 52, 13, 42, 30, 35, 26
3, 4, 6, 5// 8/ 9/ 13/ / 52, 42, 30, 35, 26
/ 4, 6, 5/ 7/ 8/ 9/ 13/ 18/ 42, 30, 35, 26/
3/ / 6, 5/ 7/ 8/ 9/ 13/ 18/ 30, 35, 26/ / 52
3/ 4/ 5/ / 7/ 8/ 9/ 13/ 18/ 26/ / 35/ 42/ 52

Время работы по сортировке списка методом быстрой сортировки зависит от упорядоченности списка. Оно будет минимальным, если на каждом шаге разбиения получаются подсписки B' и B" приблизительно равной длины, и тогда требуется около N*log2(N) шагов. Если список близок к упорядоченному, то требуется около (N*N)/2 шагов.

Рекурсивная функция quick упорядочивает участок массива s быстрой сортировкой.

```
/*              быстрая сортировка              */
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);
}
```

Здесь используются два индекса i и j, проходящие части массива навстречу друг другу (см. Рисунок - 30). При этом i всегда фиксирует разделяющий элемент cnt=s[low], слева от которого находятся числа, не большие cnt, а справа от i - числа, большие cnt. Возможны три случая: при s[i+1]<=cnt; при s[i+1]>cnt и s[j]<=cnt; при s[i+1]>cnt и s[j]>cnt. По окончании работы i=j, и cnt=s[i] устанавливается на своем месте.

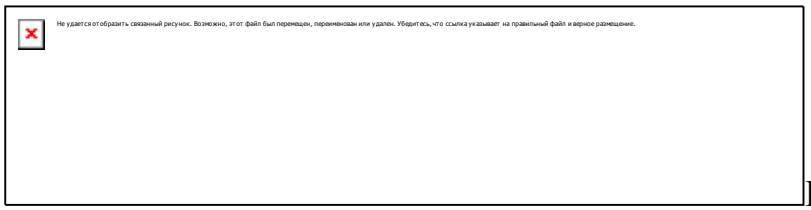


Рисунок - 30. Схема быстрой сортировки.

Быстрая сортировка требует дополнительной памяти порядка log2(N) для выполнения рекурсивной функции quick (неявный стек).

Оценка среднего количества действий, необходимых для выполнения быстрой сортировки списка из N различных чисел, получена как оценка отношения числа различных возможных последовательностей из N различных чисел, равного N!, и общего количества действий C(N), необходимых для выполнения быстрой сортировки всех различных последовательностей. Доказано, что C(N)/N! < 2*N*ln(N).

Распределяющая сортировка. Предположим, что элементы линейного списка B есть T-разрядные положительные десятичные числа D(j,n) - j-я справа цифра в десятичном числе n>=0, т.е. D(j,n)=floor(n/m)%10, где m=10^(j-1). Пусть B0,B1,...,B9 - вспомогательные списки (карманы), вначале пустые.

Для реализации распределяющей сортировки выполняется процедура, состоящая из двух процессов, называемых распределение и сборка для j=1,2,...,T.

РАСПРЕДЕЛЕНИЕ заключается в том, что элемент Ki (i=1,N) из B добавляется как последний в список Bm, где m=D(j,Ki), и таким образом получаем десять списков, в каждом из которых j-тые разряды чисел одинаковы и равны m.

СБОРКА объединяет списки B0,B1,...,B9 в этом же порядке, образуя один список B.

Рассмотрим реализацию распределяющей сортировки при T=2 для списка: B= .

РАСПРЕДЕЛЕНИЕ-1:
B0=, B1=< >, B2=, B3=, B4=,
B5=, B6=,B7=, B8=, B9=.

```
        СБОРКА-1:
B=
        РАСПРЕДЕЛЕНИЕ-2:
B0=, B1=, B2=,
B3=, B4=, B5=, B6=< >,B7=< >,B8=< >, B9=< >.
        СБОРКА-2:
B=.
```

Количество действий, необходимых для сортировки N T-цифровых чисел, определяется как Q(N*T). Недостатком этого метода является необходимость использования дополнительной памяти под карманы.

Однако можно исходный список представить как связанный и сортировку организовать так, чтобы для карманов B0,B1,...,B9 не использовать дополнительной памяти, элементы списка не перемещать, а с помощью перестановки указателей присоединять их к тому или иному карману.

В представленной ниже программе функция pocket реализует распределяющую сортировку связанного линейного списка (указатель q), в котором содержатся T-разрядные десятичные положительные числа, без использования дополнительной памяти; в функции a[i], b[i] указывают соответственно на первый и на последний элементы кармана Bi.

```
/*  вызов распределяющей сортировки списка   */
#include
#include
typedef struct str
{ long val;
  struct str *n; } SP1;
main()
{ int i;
  SP1 *q=malloc(sizeof(SP1)),*r;
  SP1 *pocket(SP1 * ,int );
  long a[14]={ 0,7,18,3,52,4,6,8,5,13,42,30,35,26 };
  q->n=NULL;
  q->val=a[0];
  r=q;
  printf(" %d",a[0]);
  for(i=1;in=malloc(sizeof(SP1));
       r->val=a[i];
       (r->n)->n=NULL;
       r=r->n;
       printf(" %d",a[i]);
    }
  r=pocket(q,2);
```

```
  printf("\n");          /*  печать результатов   */
  while (r!=NULL)
    {   printf(" %d",r->val);
        r=r->n;
    }
}
/*   распределяющая  сортировка  списка          */
SP1 *pocket(SP1 *q,int t)
{ int i,j,k,m=1;
  SP1 *r, *gg, *a[10], *b[10];
  gg=q;
  for(j=1;j<=t;j++)
    {  for(i=0;i<=9;i++) a[i]=(b[i]=NULL);
       while(q!=NULL)
         {  k=((int)(q->val/m))%(int)10;
            r=b[k];
            if (a[k]==NULL) a[k]=q;
            else r->n=q;
            r=b[k]=q;
            q=q->n;
            r->n=NULL;
         }
      for(i=0;i<=9;i++)
      if (a[i]!=NULL) break;
      q=a[i];
      r=b[i];
      for(k=i+1;k<=9;k++)
      if(a[k]!=NULL)
         {  r->n=a[k];
            r=b[k];
         }
      m=m*10;
    }
  return (gg);
}
```

На Рисунок - 31 показана схема включения очередного элемента списка в К-й карман.

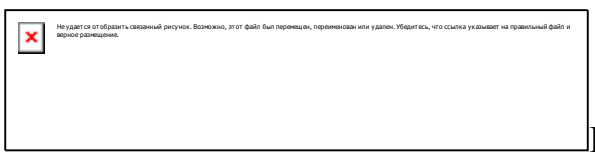


Рисунок - 31. Схема включения очередного элемента списка в карман.

Разновидностью распределяющей сортировки является битовая сортировка. В ней элементы списка интерпретируются как двоичные числа, и D(j,n) обо-

значает j-ю справа двоичную цифру числа n. При этой сортировке в процессе РАСПРЕДЕЛЕНИЕ требуются только два вспомогательных кармана B0 и B1; их можно разместить в одном массиве, двигая списки B0 и B1 навстречу друг другу и отмечая точку встречи. Для осуществления СБОРКИ нужно за списком B0 написать инвертированный список B1.

Так как выделение j-го бита требует только операций сдвига, то битовая сортировка хорошо подходит для внешней сортировки на магнитных лентах и дисках.

Разновидностью битовой сортировки является бинарная сортировка. Здесь из всех элементов списка B= выделяются его минимальный и максимальный элементы и находится их среднее арифметическое m=(MIN+MAX)/2. Список B разбивается на подсписки B1 и B2, причем в B1 попадают элементы, не большие m, а в список B2 - элементы, большие m. Потом для непустых подсписков B1 и B2 сортировка продолжается рекурсивно.

2.3.1. Последовательный поиск

Задача поиска. Пусть заданы линейные списки: список элементов B= и список ключей V= (в простейшем случае это целые числа). Требуется для каждого значения Vi из V найти множество всех совпадающих с ним элементов из B. Чаще всего встречается ситуация когда V содержит один элемент, а в B имеется не более одного такого элемента.

Эффективность некоторого алгоритма поиска A оценивается максимальным Max{A} и средним Avg{A} количествами сравнений, необходимых для нахождения элемента V в B. Если Pi - относительная частота использования элемента Ki в B, а Si - количество сравнений, необходимое для его поиска, то

$$Max\{A\} = \max\{ Si, i=1,n \} \; ; \quad Avg\{A\} = \sum_{i=1}^{n} Pi\, Si \; .$$

Последовательный поиск предусматривает последовательный просмотр всех элементов списка B в порядке их расположения, пока не найдется элемент равный V. Если достоверно неизвестно, что такой элемент имеется в списке, то необходимо следить за тем, чтобы поиск не вышел за пределы списка, что достигается использованием стоппера.

Очевидно, что Max последовательного поиска равен N. Если частота использования каждого элемента списка одинакова, т.е. P=1/N, то Avg последовательного поиска равно N/2. При различной частоте использования элементов Avg можно улучшить, если поместить часто встречаемые элементы в начало списка.

Пусть во входном потоке задано 100 целых чисел К1,К2,... К100 и ключ V. Составим программу для последовательного хранения элементов Кi и поиска среди них элемента, равного V, причем такого элемента может и не быть в списке. Без использования стоппера программа может быть реализована следующим образом:

```
/*    последовательный поиск без стоппера    */
#include
main()
{
int k[100],v,i;
for (i=0;i
```

С использованием стоппера программу можно записать в виде:

```
/*  последовательный поиск со стоппером    */
#include
main()
{
int k[101],v,i;
for (i=0;i
```

2.3.2. Бинарный поиск

Для упорядоченных линейных списков существуют более эффективные алгоритмы поиска, хотя и для таких списков применим последовательный поиск. Бинарный поиск состоит в том, что ключ V сравнивается со средним элементом списка. Если эти значения окажутся равными, то искомый элемент найден, в противном случае поиск продолжается в одной из половин списка.

Нахождение элемента бинарным поиском осуществляется очень быстро. Max бинарного поиска равен log2(N), и при одинаковой частоте использования каждого элемента Avg бинарного поиска равен log2(N). Недостаток бинарного поиска заключается в необходимости последовательного хранения списка, что усложняет операции добавления и исключения элементов .

Пусть, например, во входном потоке задано 101 число, К1,К2,...,К100, V - элементы списка и ключ. Известно, что список упорядочен по возрастанию, и элемент V в списке имеется. Составим программу для ввода данных и осуществления бинарного поиска ключа V в списке К1,К2,...,К100.

```
/*    Бинарный поиск      */
#include
```

```
main()
{
int k[100],v,i,j,m;
for (i=0;i
```

2.3.3. M-блочный поиск

Этот способ удобен при индексном хранении списка. Предполагается, что исходный упорядоченный список B длины N разбит на M подсписков B1,B2,...,Bm длины N1,N2,...,Nm, таким образом, что B=B1,B2,..,Bm.

Для нахождения ключа V, нужно сначала определить первый из списков Bi, i=1,M, последний элемент которого больше V, а потом применить последовательный поиск к списку Bi.

Хранение списков Bi может быть связным или последовательным. Если длины всех подсписков приблизительно равны и M= N, то Max M-блочного поиска равен 2 N. При одинаковой частоте использования элементов Avg M-блочного поиска равен N.

Описанный алгоритм усложняется, если не известно, действительно ли в списке имеется элемент, совпадающий с ключом V. При этом возможны случаи: либо такого элемента в списке нет, либо их несколько.

Если вместо ключа V имеется упорядоченный список ключей, то последовательный или M-блочный поиск может оказаться более удобным, чем бинарный, поскольку не требуется повторной инициализации для каждого нового ключа из списка V.

2.3.4. Методы вычисления адреса

Методы вычисления адреса. Пусть в каждом из M элементов массива T содержится элемент списка (например целое положительное число). Если имеется некоторая функция H(V), вычисляющая однозначно по элементу V его адрес - целое положительное число из интервала [0,M-1],то V можно хранить в массиве T с номером H(V) т.е. V=T(H(V)). При таком хранении поиск любого элемента происходит за постоянное время не зависящее от M.

Массив T называется массивом хеширования, а функция H - функцией хеширования.

При конкретном применении хеширования обычно имеется определенная область возможных значений элементов списка V и некоторая информация о них. На основе этого выбирается размер массива хеширования M и строится функция хеширования. Критерием для выбора M и H является возможность их эффективного использования.

Пусть нужно хранить линейный список из элементов K1,K2,..,Kn, таких, что при Ki=Kj, mod(Ki,26)= mod(Kj,26). Для хранения списка выберем массив хеширования T(26) с пространством адресов 0-25 и функцию хеширования H(V)= mod(V,26). Массив T заполняется элементами T(H(Ki))=Ki и T(j)=0 если j (H(K1), H(K2),..,H(Kn)).

Поиск элемента V в массиве T с присваиванием Z его индекса если V содержится в T, или -1, если V не содержится в T, осуществляется следующим образом

```
int t[26],v,z,i;
i=(int)fmod((double)v,26.0);
if(t[i]==v) z=i;
else z=-1;
```

Добавление нового элемента V в список с возвращением в Z индекса элемента, где он будет храниться, реализуется фрагментом

```
z=(int)fmod((doule)v,26.0);
t[z]=v;
```

а исключение элемента V из списка присваиванием

```
t[(int)fmod((double)v,26)]=0;
```

Теперь рассмотрим более сложный случай, когда условие Ki=Kj H(Ki)=H(Kj) не выполняется. Пусть V - множество возможных элементов списка (целые положительные числа), в котором максимальное число элементов равно 6. Возьмем M=8 и в качестве функции хеширования выберем функцию H(V)=Mod(V,8).

Предположим, что B=, причем H(K1)=5, H(K2)=3, H(K3)=6, H(K4)=3, H(K5)=1, т.е. H(K2)=H(K4) хотя K2=K4. Такая ситуация называется коллизией, и в этом случае при заполнении массива хеширования требуется метод для ее разрешения. Обычно выбирается первая свободная ячейка за собственным адресом. Для нашего случая массив T[8] может иметь вид

T= .

При наличии коллизий усложняются все алгоритмы работы с массивом хеширования. Рассмотрим работу с массивом T[100], т.е. с пространством адресов от 0 до 99. Пусть количество элементов N не более 99, тогда в T всегда

будет хотя бы один свободный элемент равный нулю. Для объявления массива используем оператор

```
int static t[100];
```

Добавление в массив T нового элемента Z с занесением его адреса в I и числа элементов в N выполняется так:

```
i=h(z);
while (t[i]!=0 && t[i]!=z)
if (i==99) i=0;
else i++;
if (t[i]!=z)  t[i]=z,  n++;
```

Поиск в массиве T элемента Z с присвоением I индекса Z, если Z имеется в T, или I=-1, если такого элемента нет, реализуется следующим образом:

```
i=h(z);
while (t[i]!=0 && t[i]!=z)
if (i==99) i=0;
else i++;
if (t[i]==0) i=-1;
```

При наличии коллизий исключение элемента из списка путем пометки его как пустого, т.е. t[i]=0, может привести к ошибке. Например, если из списка B исключить элемент K2, то получим массив хеширования в виде T=, в котором невозможно найти элемент K4, поскольку H(K4)=3, а T(3)=0. В таких случаях при исключении элемента из списка можно записывать в массив хеширования некоторое значение непринадлежащее области значений элементов списка и не равное нулю. При работе с таким массивом это значение будет указывать на то, что нужно просматривать со средние ячейки.

Достоинство методов вычисления адреса состоит в том, что они самые быстрые, а недостаток в том, что порядок элементов в массиве T не совпадает с их порядком в списке, кроме того довольно сложно осуществить динамическое расширение массива T.

2.3.5. Выбор в линейных списках

Задача выбора. Задан линейный список целых, различных по значению чисел B=, требуется найти элемент, имеющий i-тое наибольшее значение в порядке убывания элементов. При i=1 задача эквивалентна поиску максимального элемента, при i=2 поиску элемента с вторым наибольшим значением.

Поставленная задача может быть получена из задачи поиска j-того минимального значения заменой i=n-j+1 и поиском i-того максимального значения. Особый интерес представляет задача выбора при i=a/n, 0<a<1, в частности, задача выбора медианы при a=1/2.

Все варианты задачи выбора легко решаются, если список B полностью отсортирован, тогда просто нужно выбрать i-тый элемент. Однако в результате полной сортировки списка B получается больше информации, чем требуется для решения поставленной задачи.

Количество действий можно уменьшить применяя сортировку выбором только частично до i-того элемента. Это можно сделать, напри мер при помощи функции findi

```
/*  выбор путем частичной сортировки  */
int findi(int *s, int n, int i)
{
  int c,j,k;
  for (k=0; k<=i; k++)
  for (j=k+1; j<=n; j++)
  if (s[k] < s[j])
  {  c=s[k];
     s[k]=s[j];
     s[j]=c;
  }
  return s[i];
}
```

Эта функция ищет элемент с индексом i, частично сортируя массив s, и выполняет при этом (n*i) сравнений. Отсюда следует, что функция findi приемлима для решения задачи при малом значении i, и малоэффективна при нахождении медианы.

Для решения задачи выбора i-того наибольшего значения в списке B модифицируем алгоритм быстрой сортировки. Список B разбиваем элементом K1 на подсписки B' и B", такие, что если Ki -B', то Ki>K1, и если Ki - B", то Ki<K1, и список B реорганизуется в список B',K1,B". Если K1 элемент располагается в списке на j-том месте и j=i, то искомый элемент найден. При j>i наибольшее значение ищется в списке B'; при j<i будем искать (i-j) значение в подсписке B".

Алгоритм выбора на базе быстрой сортировки в общем эффективен, но для улучшения алгоритма необходимо, чтобы разбиение списка на подсписки осуществлялось почти пополам. Следующий алгоритм эффективно решает

задачу выбора i-того наибольшего элемента в списке B, деля его на подсписки примерно равной величины.

1. Если N<21, то выбрать i-тый наибольший элемент списка B обычной сортировкой.

2. Если N>21 разделим список на P=N/7 подсписков по 7 элементов в каждом, кроме последнего в котором mod(N,7) элементов.

3. Определим список W из медиан полученных подсписков (четвертых наибольших значений) и найдем в W его медиану M (рекурсивно при помощи данного алгоритма) т.е. (P/2+1)-й наибольший элемент.

4. С помощью элемента M разобьем список B на два подсписка B' с j элементами большими или равными M, и B" с N-j элементами меньшими M. При j>i повторим процедуру поиска сначала, но только в подсписке B'. При j=i искомый элемент найден, равен M и поиск прекращается. При j < i будем искать (i-j)-тый наибольший элемент в списке B".

```
/*   алгоритм выбора делением списка почти пополам   */
int search (int *b, int n, int i)
{
 int findi(int  *, int, int);
 int t, m, j, p, s, *w;
 if (n=m) j++;
   if (j>i)
   {
     for (p=0, t=0; p < n; t++)
     if (b[t]>=m)
     { b[p]=b[t]; p++; }
     m=search(b, j, i);            /*   поиск в B"    */
   }
   if (j < i)
   {
     for (p=0, t=0; t < n; t++)
       if (b[t] < m)     b[p++]=b[t];
       m=search(b, n-j, i-j);       /*   поиск в B"    */
   }
 return m;
}
```

Рекурсивная функция search реализует алгоритм выбора i-того наибольшего значения. Для ее вызова можно использовать следующую программу

```
#include
```

```
#include
main()
{
  int search (int *b, int n, int i);
  int *b;
  int l, i, k, t;
  scanf("%d%d",&l,&i);
  printf
  ("\nВыбор %d максимального элемента из %d штук",i,l);
  b=(int *)(calloc(100,sizeof(int)));
  for (k=0; k
```

Используя метод математической индукции, можно доказать, что для функции search требуется выполнить в самом неблагоприятном случае 28*N сравнений.

Действительно, если N<21, то выполнение функции findi потребует сравнений порядка N*(N-1)/2, т.е. меньше чем 28*N. Предположим, что для любого T<N количество сравнений при выполнении функции search не более 28*T и подсчитаем, сколько сравнений потребуется функции search при произвольном значении N. Для поиска медианы в каждом из подсписков функцией findi требуется не более 7*(7-1)/2=21 сравнений, а для формирования массива W в целом не более 21*(N/7)=3*N сравнений. По предположению индукции для поиска медианы в массиве W длины N/7 требуется 28*(N/7)=4*N сравнений. После удаления из B части элементов с помощью медианы в B' (или в B") останется не более N*5/7 элементов, и для удаления ненужных элементов необходимо количество сравнений порядка N. Для поиска в оставшейся части массива (в B' или B") по предположению индукции требуется не более 28*(N*5/7)=20*N сравнений. Таким образом, всего потребуется 3*N+4*N+N+20*N=28*N сравнений, т.е. выдвинутое предположение доказано.

ЛИТЕРАТУРА

1. Подбельский В.В., Фомин С.С. Программирование на языке С: Учеб. пособие. – М.: Финансы и статистика, 2002 г. – 600 с.
2. Подбельский В.В. Язык С++: Учеб. пособие. – М.: Финансы и статистика, 2002 г. – 560 с.
3. Тихомиров Ю. Visual C++ 6. – СПб.: БХВ-Санкт-Петербург, 1998. – 465 с.
4. Любимский Э.З., Мартынюк В.В., Трифонов Н.П. Программирование. – М.: Наука, 1980. – 607 с.
5. Керниган Б., Ритчи Д. Язык программирования Си. – М.: Финансы и статистика, 1992. - 272 с.
6. Романовская Л.М., Русс Т.В., Свитковский С.Г. Программирование в среде СИ для ПЭВМ ЕС. – М.: Финансы и статистика, 1992. – 352 с.
7. Прата С. Язык программирования С++. Лекции и упражнения, Учебник. – К.: Издательство «ДиаСофт», 2001. – 656 с.
8. Дейтел Х., Дейтел П. Как программировать на С++. – М.: ЗАО «Издательство БИНОМ», 2001. – 1152 с.
9. Г.Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на С++. – М.: «Издательство Бином», СПб.: «Невский диалект», 1999. – 560 с.