Національний технічний університет України

“Київський політехнічний інститут”

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

ПРОГРАМУВАННЯ

Конспект лекцій

Частина 1

“Алгоритмічне програмування”

Київ 2014

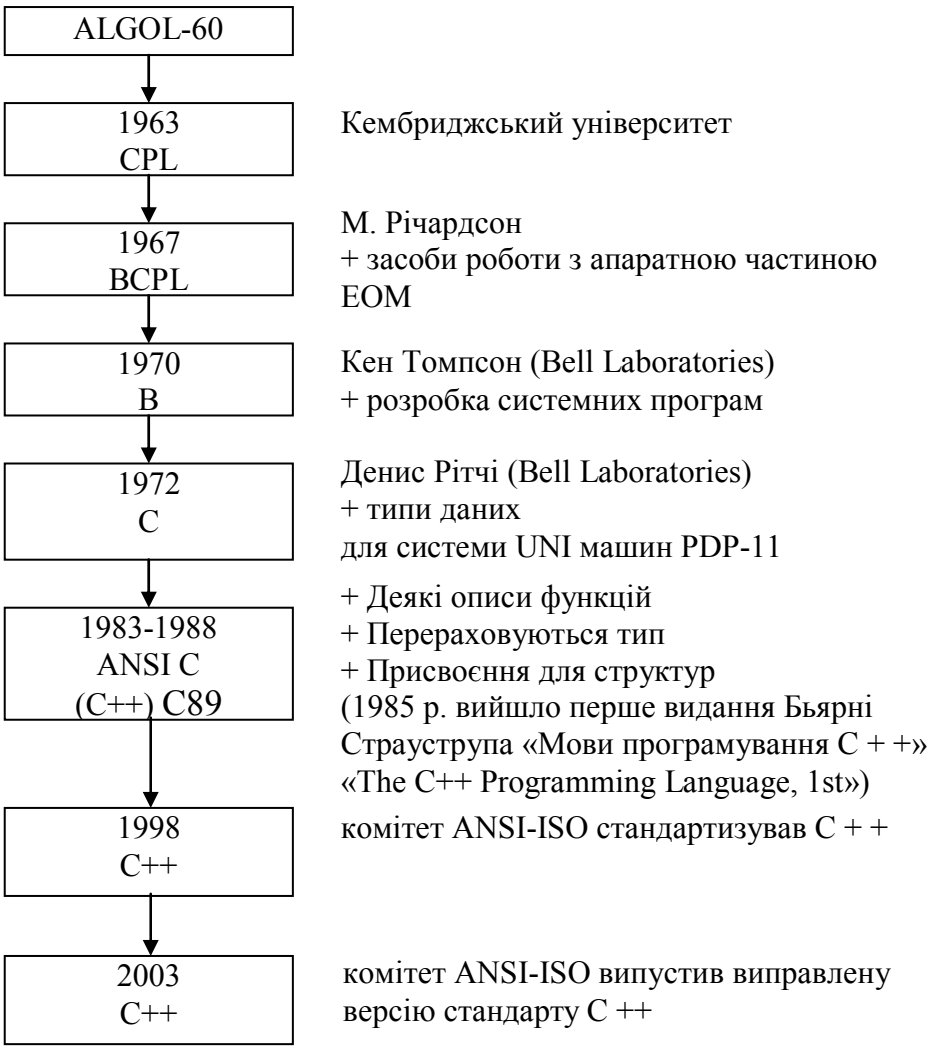
Лекція 1

РОЗДІЛ 1. ПРОСТІ ЕЛЕМЕНТИ МОВИ

1.1. Особливості мови С

ВВЕДЕННЯ

Мова програмування С створена на фірмі Bell Laboratories в 1972р. Денісом Рітчі (*Dennis MacAlistair Ritchie*). Попередником цієї мови є мова АЛГОЛ-60, на основі якої була розроблена мова CPL (мова комбінованого програмування) в 1963р. (Кембриджський і Лондонський університети) потім BCPL (базова мова комбінованого програмування) 1967р. (Кембриджський університет). Метою цих розробок було відродити мову АЛГОЛ, зберегти контакт із ЕОМ. Тобто поруч із можливостями універсальної процедурної мови були додані засоби роботи з апаратною частиною ЕОМ. Тому мова BCPL виявилася громіздким і в 1970р. Кеном Томпсоном (*Kenneth Lane Thompson*) в Bell Laboratories була розроблена мова B, яка була вузько спрямована на розробку системних програм.



Ця мова й стала основою для створення мови С. Основна заслуга Д. Рітчі - це додавання вдалої системи типів даних.

Початкове призначення мови С - системне програмування . Сама мова була невід'ємною частиною системи UNI для ЕОМ типу PDP-11. Так приблизно з 13 тис. рядків цієї системи

лише 800 були написані на асемблері, інші на С.
Можна відзначити такі особливості мови:
1. Мова С належить до мов зі слабкою типізацією даних. Набагато ширше, ніж у мові Паскаль, передбачене неявне перетворення типів.
2. Передбачене використання покажчиків, які дають можливість оперувати з адресами й непрямою адресацією.
3. Клас об'єктів мови обмежений. Виключені такі структури даних як логічні, множини.
Оскільки мова С тісно пов'язана з операційною системою UNI, то багато можливостей реалізуються функціями операційної системи. Це забезпечило компактність і високу ефективність даної мови.

Мова відзначається внутрішньою єдністю, що властива мовам "для людини" (Паскаль, Лисп). Має невеликий перелік вихідних засобів, з яких можна створювати досить складні конструкції.
Реалізує принципи структурного програмування, зокрема , блокову побудову.
Усе це сприяло широкому поширенню мови не тільки для системних цілей, але й для розв'язку широкого класу прикладних завдань.
Незважаючи на те, що С пов'язана із системою UNI PDP-11, відзначені позитивні властивості сприяли її широкому застосуванню на інших ЕОМ і операційних системах. Тому говорять про її високу мобільність і доброї переносимості..

Однак перші варіанти мови не були уніфіковані, через що питання перенесення все-таки виникали. Тому в 1983р. інститут американських національних стандартів (ANSI) створив комітет для розробки сучасної машинно незалежної мови С. І в 1988р. ця робота закінчилася створенням мови "ANSI C". ця версія широко використовується на персональних ЕОМ, зберігає більшість властивостей вихідної мови, відрізняється:

1. деяким описом функцій,
2. наявністю типу, що перелічується,
3. дозволяє операцію присвоєння для структур (записів), розширеною й уніфікованою бібліотекою функцій.

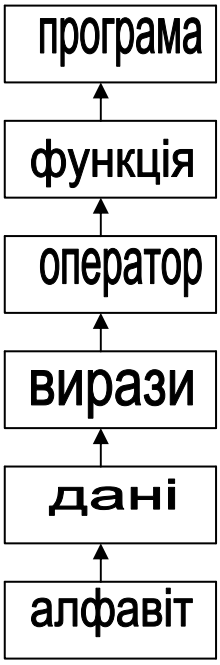
Можна вважати, що мова С, з одного боку, є зручної, виразної й гнучкої для програмування широкого класу завдань. З іншого боку - вона в достатній мірі наближена до ЕОМ, дає засоби контролю процесом реалізації програми, але зберігає певну дистанцію, яка дозволяє не враховувати всі особливості архітектури ЕОМ. Можна вважати, що це мова відносно низького рівня.

1985 р. вийшло перше видання Бьярні Страуструпа «Мови програмування С + +»
«The C++ Programming Language, 1st»

Говорять, що мова С призначена для професіоналів- програмістів, а не для початківців. Це створює певні труднощі при вивченні мови.

ОСНОВНІ ПОНЯТТЯ МОВИ

Структура мови С++



Спочатку розглянемо деяку мінімальну підмножину мови, яка дозволить писати прості програми.

Алфавіт

Символи мови складаються із трьох груп: **літер, цифр, спеціальних символів**.
Літери – це букви латинського алфавіту (A-Z, a-z), а також кирилиці, які використовуються в коментарях і рядкових константах (А-я).

Спецсимволи складаються з:

1) знаків арифметичних операцій (+ додавання; - віднімання; * множення; /ділення, остача від ділення x % y цілих чисел);
2) знаків відношень;
3) роздільників;
4) службових слів.

1. Арифметичні операції:

Додавання	total = var1 + var2
Віднімання	total = var1 - var2
Множення	total = var1 * var2
Ділення	total = var1 / var2

	Оператор	total =
+	збільшення	total + 1
	Оператор	total =
-	зменшення	total - 1

(++)(--)variable - *префіксним оператором збільшення*;

```
int A = 7, B=5;
cout << "начальное значение A равно" << A <<
A += ++B;
cout << "конечное значение A равно" << A << endl;
```

?

1. B=B+1
2. A=A+B
3. A=13

variable(++)(--) - *постфіксним оператором збільшення*;

```
int A = 7, B=5;
cout << "начальное значение A равно" << A <<
A +=B++;
cout << "конечное значение A равно" << A << endl;
```

?

1. A=A+B
2. B=B+1
3. A=12

Операція	Функція
%	Взяття по модулю або залишок; повертає залишок цілочисленного розподілу
~	Додаток; інвертує біти значення
&	Побітове І (також &X – взяти адрес X)
|	Побітове включающе АБО
^	Побітовое виключающе АБО
<<	Зсув вліво; зсуває біти значення вліво на зазначену кількість розрядів
>>	Зсув вправо; зсуває біти значення вправо на зазначену кількість розрядів

Операції піднесення до степеня не передбачено.

2. Знаки відносин і логічних операцій:

>, <	Більше, менше
==	Дорівнює (2 знака рівності)
!=	Не дорівнює
&&	Логічне І
||	Логічне АБО
!	Логічне НЕ

3. Роздільники: . , ( ) [ ] { } ' " = : ;

Фігурні дужки мають значення операторных дужок (Begin - End мови Паскаль). Для коментарів використовуються пари знаків /*КОМЕНТАР*/.

4. Службові слова:

auto	break	case	char	continue	default	do	double
else	entry	enum	extern	float	for	goto	if
int	long	register	return	short	sizeof	static	struct
switch	typedef	union	unsigned	void	while		

Всього 30 службових слів від auto до while.

Константи

Константами називаються перерахування величин у програмі.
Розділяють константи таких типів:
1. цілі (243)
2. з плаваючою крапкою (дійсні) (543.8)
3. символьні ('A') - 1 байт
4. строкові ("A") - 2 байта A + NULL
5. переліковного типу (ANSI-C).

Цілі константи можуть записуватися в десятковій, восьмеричній і шістнадцятиричній системах вирахування.
Десяткова ціла константа подається звичайним образом (зі знаком або без нього): -2561; 458.

Ознакою восьмеричної константи є провідний нуль ліворуч: Ø257.

Шістнадцятирічна константа визначається двома початковими символами: ØX.

Нагадуємо, що для вистави чисел від 10 до 15 у цій системі використовуються латинські букви: A - 10, B - 11, C - 12, D - 13, E - 14, F - 15.

Тому 31_{10} в цій системі буде записано ØX1F (1*16+ F=31).

Крім звичайних цілих констант можна використовувати цілі подвійної точності, які в пам'яті займають 2 слова.

До них приєднується літера L: 371L. Діапазон: -2.147.483.648<=2.147.483.647.

Константи з плаваючою крапкою (floating point) можуть подаватися у формі з фіксованою крапкою: 561.25, .5, 100.

При цьому в пам'яті розміщаються звичайним способом (2 слова).
Якщо ж константа задається у формі із плаваючою крапкою (експонентна форма)1E-06, 1.8E4, то в деяких трансляторах вона автоматично подається з подвійною точністю, тобто в 4 слова

Символьні константи

Символьні константи (CHAR) набувають значення одного символу й подаються в апострофах 'x', 'a', 'r' і займають 1 байт.
Недруковані символи, які не мають графічного зображення, подаються умовно двома символами: перший - використовується зворотний слеш, а другий - який-небудь певний символ. Так звана зворотна послідовність перемикання коду (escape sequence або ескейп послідовність).

Перехід на наступну строку	\n'
Горизонтальна табуляція	\t\'
Перехід на попередню позицію	\b'
Переведення керетки	\r'
Перехід на наступну сторінку	\f'
Апостроф	\"
Звуковий сигнал	

	\a'
Вертикальна табуляція	\v'

Строковыі константи

Строковыі константи (строки, string) це послідовність символів, обмежена подвійними лапками: "string". Для переносу на інший рядок використовується зворотний слеш

"морозный\
день" - 12+1 байт

Як і в мові Паскаль, константи можуть задаватися своїм іменем. Але в мові С немає спеціального розділу опису констант.

Тому визначення констант реалізується 3-а способами:

1. Процесором і має вигляд:

#define<им'я константи> <літерал або значення>
#define<им'я константи ><вираз з констант>

Наприглад:

```
#define LN 50
#define PI 3.141592
```

Наприклад, наступний оператор створює макрокоманду CUBE:

```
#define CUBE(x) ((x)*(x)*(x))
```

або

Наприклад, наступний оператор створює макрокоманду DELAY:

```
#define delay(x)
{ \
printf( "Задержка на %d", x); \
for (long int i=0; i < x; i++) \
; \
}
```

- ознака звертання до процесора. Для зручності константи позначаються більшими буквами. Перед трансляцією процесор скрізь у тілі програми заміняє ім'я константи її значенням.

2. За допомогою слова const: **const [тип] <имя> = <значення>**

```
const float  pi  = 3.1415926;
const  maxint = 32767;
```

3. Оголошення перерахування починається із ключового слова enum і має два формати вистави:

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

Наприклад :

```
enum days { sun, mon, tues, wed, thur, fri, }  anyday;
enum day {sat, sun}  weekend, superday;
```

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

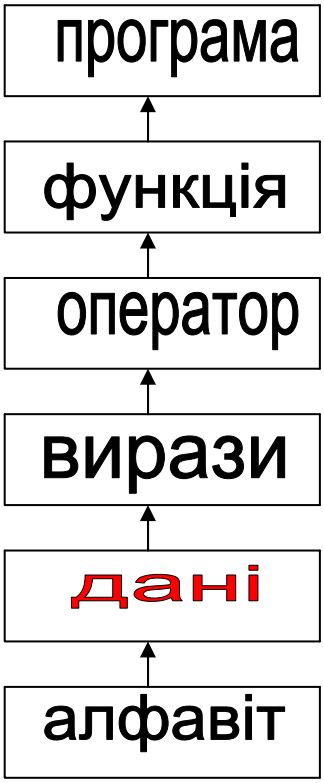
Наприклад :

```
anyday = mon; // можно
 anyday = 1;// нельзя, хотя mon ==1
```

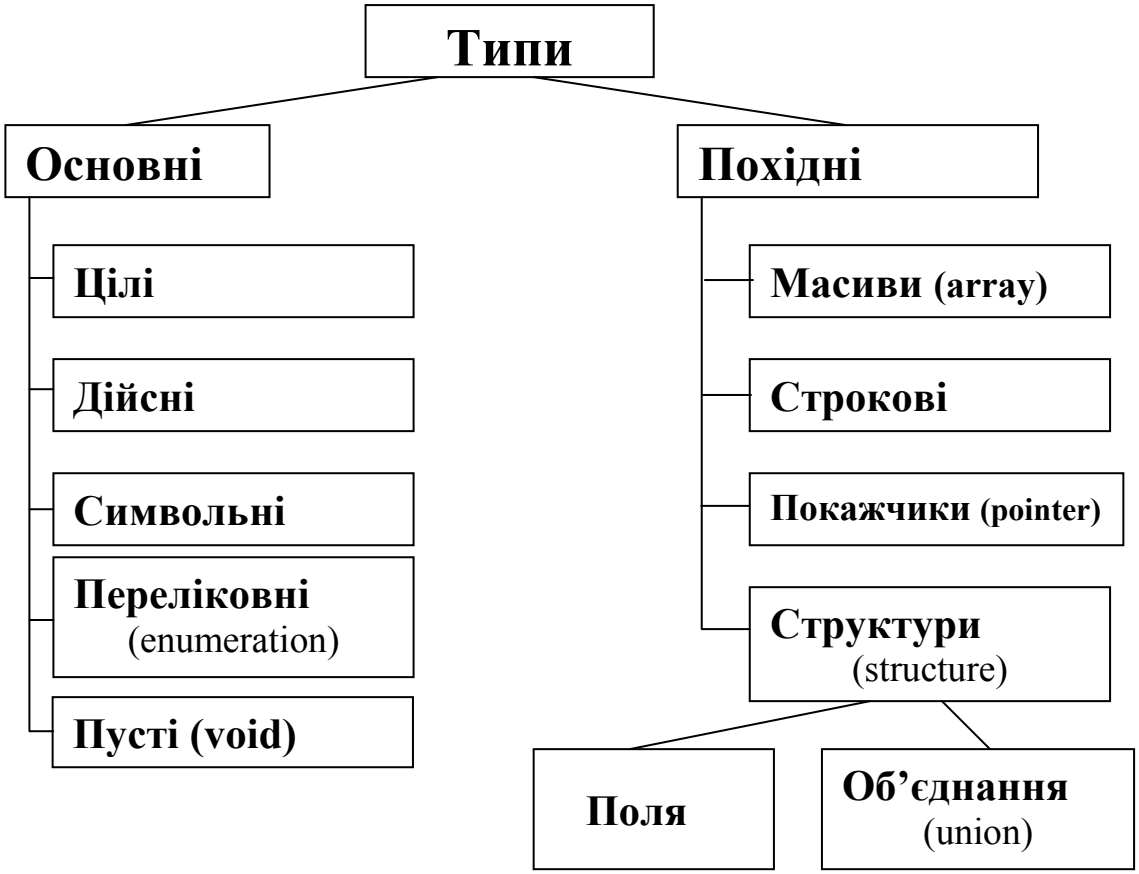
Лекція 2

1.2 Структура даних і вирази

Типи і змінні



Змінні в мові С діляться на дві групи типів: основні й похідні.



До основних належать ***5 типів***:

1. Цілі;

2. Дійсні (з плаваючою крапкою);

3. Символьні (char);

4. Переліковні (enumeration);

5. Порожні (void).

З основних типів створюються похідні ***4- х типів***:

1. Масиви (array);

2. Покажчики (pointer);

3. Строкові (Рядки);

4. Структури або записи (structure) і їх різновиди поля й об'єднання (union);

Цілі

Як ми вже відзначали, крім стандартних цілих можна ще використовувати й деякі інші різновиди: short - короткі; long - довгі; unsigned - без знака.
Оголошення цілих змінних має такий вид:

Тип	Розмір	Діапазон
char	1	0 ÷ 256
int	2	-32768 ÷ 32767
short	2	-32768 до 32767
long	4	-2 147 483 648 ÷ 2 147 483 647
unsigned shar	1	0 ÷ 255
unsigned int	2	0 ÷ 65535
unsigned short	2	0 ÷ 65535
unsigned long	4	0 ÷ 4 294 967 295

```
int             i, n;
short int       low, hight;
long int        max;
unsigned int    kl;
```

В останніх трьох описах ключове слово int може бути пропущене. ЗАЛЕЖНО від транслятора short int можуть займати 1 байт або 2 байта.

В описах змінних можна задавати й початкове значення

```
long max = 32767L;
short dogs, cats = 93;
```

В останньому випадку змінна cats має значення 93, а змінна dogs - ніякого.

Дійсні

С плаваючою крапкою

Опис таких змінних має вигляд:

Тип	Розмір	Діапазон	Точність
float	2 слова - 4 байта - 32 біта	10e-38 ÷ 10e38	5 знаків
double	4 слова - 8 байт - 64 біта	10e-308 ÷ 10e308	15 знаків
long double	5 слів - 10 байт - 80 біт	10e-4932 ÷ 10e4932	19 знаків

Наприклад:

```
float pi_float;
double pi_double;
long double pi_long_double;

pi_float = 3.1415;
pi_double = 3.14159265358979;
pi_long_double = 3.141592653589793238;
```

Символьні змінні

Задаються описом:

```
char c, ch='Y';
char esc ='\x1B';
```

Управляюча послідовність	Найменування	Шістнадцятирічний формат
\a	Звуковий сигнал	007
\b	Повернення на крок	008
\t	Горизонтальна табуляція	009
\n	Перехід на нову строку	00A
\v	Вертикальна табуляція	00B
\r	Повернення каретки	00C
\f	Перевод формата	00D
\"	Лапки	022
\'	Апостроф	027
\0	Ноль-символ	000
\\	Зворотня дробова риса	05C
\0ddd	Символ набору кодів в вісімковому представленні	
\xddd	набір кодів в шістнадцятковому представленні	

Як і для цілих змінних, початкові значення змінних інших типів також можуть бути задані в описах.

Переліковний тип - ANSI - C

Переліковний тип (enum) використовується для опису об'єктів з певним набором, наприклад {winter, spring, summer, autumn}. Тоді можна записати таке оголошення (декларацію) :

```
enum seasons {wint, spring, sum, autumn}
```

І описати відповідні змінні:

```
enum seasons a, b, c.
```

Кожна із зазначених змінних може здобувати одне із чотирьох значень. І оголошення типу, і опис змінних можуть бути об'єднані:

```
enum seasons {wint, spring, sum, autumn} a, b, c;
```

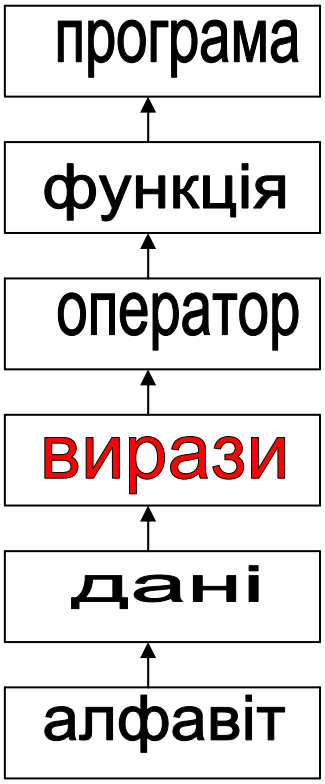
Імена, які зазначені в дужках, це не рядок. Усередині ЕОМ вони набувають значення цілих чисел: перше - 0, друге - 1, і т.д. Можна присвоїти й інші значення, але за збільшенням:

```
enum month {JAN = 1, FEB, MAR, ..., DEC};
enum days {mon = 5, tues = 8, wed = 10, thur, fri, sat, sun}.
```

Переліковний тип також можна використовувати для надання константам імен:

```
enum escapes {BELL = '\a', BACKPASE = '\b', TAB = '\t', NEWLINE = '\n', VTAB = '\V', RETURN = '\r'}.
```

Арифметичні вирази та операції присвоєння



Операції

У мові С передбачене майже 40 операцій, але розглянемо спочатку ті, які найчастіше вживаються - арифметичні операції, операцію присвоєння, зменшення й збільшення, обчислення модуля.

Арифметичні вирази та операції присвоєння

Операція присвоєння найчастіше використовується в програмах і має вигляд:

a = b, де a - певна змінна, b - вираз. Результат виразу b присвоюється змінної a. Можна використовувати й багаторазове присвоєння
a = b = c = d = 1.

Таке присвоєння виконується ⇐ праворуч ліворуч.
Якщо b є арифметичним виразом загального виду, то його результат залежить від прийнятої послідовності виконання операцій - їх старшинства. Враховуючи велику кількість операцій, існують таблиці, де обумовлений порядок виконання й старшинство.

Ієрархія операцій в С++

Операція	Ім'я	Приклад
::	Діапазон області визначення	classname::classmember_name
::	Глобальний діапазон	::variable_name
.	Вибір елемента	object.member_name
->	Вибір елемента	pointer->membername
[]	Індексація	pointer[element]
()	Виклик функції	expression(parameters)
()	Будування значення	type(parameters)
sizeof	Розмір об'єкта	sizeof expression
sizeof	Розмір типа	sizeof(type)
++	Збільшення після	variable++
++	Збільшення до	++variable
--	Зменшення після	variable--
--	Зменшення до	-- variable
&	Адрес об'єкта	&variable
*	Розйменування	*pointer
new	Створення (розміщення)	new type
delete	Видалення	delete pointer
delete[]	Видалення масива	delete pointer
~	Доповнення	~expression
!	Логічне НІ	! expression
+	Унарный плюс	+1
-	Унарный мінус	-1
()	Приведення	(type) expression
.*	Вибір елемента покажчика	object.*pointer
->	Вибір елемента покажчика	object->*pointer
*	Множення	expression * expression
/	Ділення	expression / expression
%	Взяття по модулю	expression % expression
+	Додавання (плюс)	expression + expression
-	Віднімання (мінус)	expression expression

Для зміни порядку виконання операцій у виразах використовуються круглі дужки. Проте тут є особливості.
Наприклад:

c = (d = 1) + 2; Тут компактно записано d =1; c = d + 2;
Ще один приклад a = (b = c / (d * e)) + f.
Такі записи можливі тому, що присвоєння в мові С розглядається не як оператор, а як операція.

Наприклад, операцію присвоєння i=i+2 можна записати i+=2. І взагалі, для багатьох бінарних операцій можливі операції виду op= , де op - операція. Тому діє правило e1 op=e2, де e1**,** e2 – вирази, а op - операція означає **e1=(e1) op (e2)**

Тут дужки необхідні. Наприклад:

y*=z+1 еквівалентно **y=y*(z+1)**, а не **y=y*z+1**.

Знаки операцій збільшення й зменшення можуть стояти як ліворуч, так і праворуч від операнда **i++** або **++i**. Для одного операнда це рівнозначно. Але в операціях присвоєння при цьому з'являються певні відмінності. Наприклад:

a=++b і **a=b++** це не тотожність.

1. b++ a=b
2. a=b b++

Операції збільшення й зменшення можна застосовувати лише до цілих змінних, а не до виражень **(a+b)++** - НЕ МОЖНА.

В арифметичних вираженнях операція розподілу виконується з урахуванням формату змінних.

Для цілих і символьних величин існує операція залишку від розподілу %:

5%3 буде 2. Відзначимо, що % **не можна** використовувати для чисел із плаваючою крапкою.

Особливості виконання арифметичних виразів і операції присвоєння.

В операціях присвоєння із правої сторони у виразах можуть траплятися операнди різних типів і тип результату виразу може не збігатися з типом змінної з лівої сторони. Тому виникає питання, а яким буде остаточний результат?

Послідовність дій наступна:

1. Спочатку обчислюється вираз праворуч;

2. Остаточний результат визначається змінною ліворуч.

Для визначення результату виражень діють такі правила:

1. Якщо у вираженні операнди тільки одного типу, то результат має той же самий тип;

2. Якщо операнди різного типу, то результат має "вищий тип".

Тобто у мові С типи даних мають певне ієрархію;

символьні (char)< цілі (int) < довгі (long int) < дійсні (float) < подвійної точності (double) < довгої подвійної точності (long double).

Це означає, що перед виконанням вираре всі операндb перетворюються в старший тип, тобто відбувається підвищення типу. Після обчислення результату вираження відбувається перетворення типів при виконання операцій присвоєння. І яким би не був результат вираре, він перетворюється до типу зміною ліворуч. Відбувається начебто зниження типу. Наприклад:

```
char c1, c2,c3;
int i1, i2, i3;
float f1, f2, f3;
c1='x'; // нема перетворень
c2=1000; // ціла відсікається до символу
c3=6.02e23; // плаваюча відсікається до символу
i3='x'; // символ розширюється до цілого
i3=6.02e23; // плаваюча відсікається до цілого
```

Розглянемо c2=1000; 100010= $17508=1111101000_2=1508=64+40=104_{10}$

Це код літери h.
Як бачимо, у деяких випадках, особливо при участі дійсних чисел одержуємо безглузді результати. Проте цей засіб можна використовувати для перетворення типів.

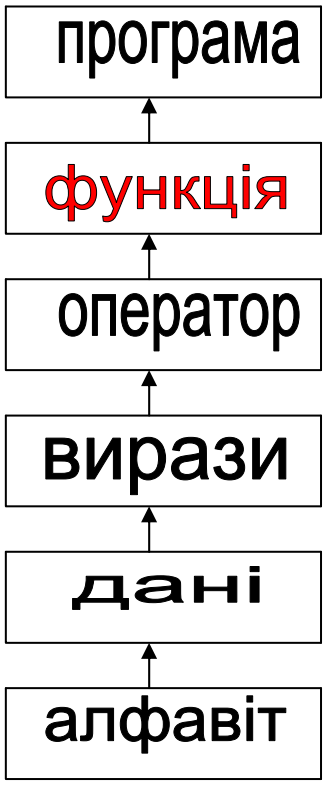
Відзначимо, що з погляду мови Паскаль ці присвоєння взагалі на можуть бути неможливі. Проте в мові С++ вони припустимі, але слід бути обережним при їхньому використанні. Крім неявного перетворення типів можна використовувати функцію явного перетворення, яка має вигляд:

(тип) <вираз>
наприклад:

```
int t1,t2;
float f1,f2=3.6;
t1=f2;// t1=
t2=(int) 1.6+7; // t2=8
f1=(int) 1.6; // f1=1.0
```

Лекція 3

Структура і приклад програми



Програма мовою С++ складається з окремих функцій. Кожна функція включає заголовок і тіло.

Тип_функції імя_функції (тип змінна, тип змінна, ...)

{

Тіло функції;

[Return (вираз);]

}

Де **тип змінна**– список формальних параметрів.

Заголовок функції - це ім'я й у круглих дужках список формальних параметрів. Поряд із усіма іншими функціями програма обов'язково повинна мати головну функцію main ().

Тому будемо завжди починати з головної функції **main ()**. По всьому тексту програми можна розміщати коментарі:

/* обчислення кореня */ - блоковий коментар або // закінчення циклу - рядковий коментар

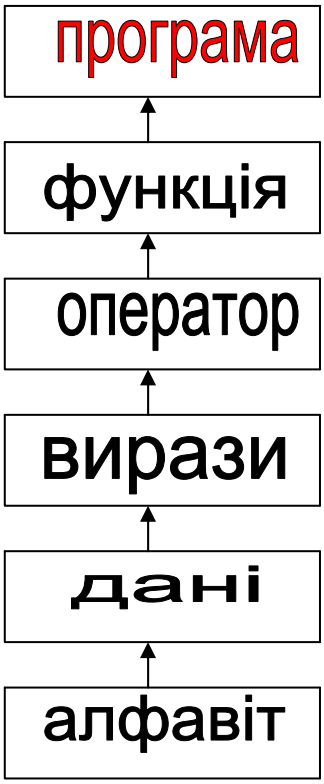
Тіло функції завжди починається й закінчується операторними дужками { }, які є аналогом BEGIN і END у мові Паскаль.

Оскільки складений оператор це є також блок, те фігурні дужки використовуються й для складених операторів.

Тіло функції складається з окремих операторів, кожний з яких обов'язково закінчується роздільником ";", незалежно від того, що за ним іде далі. Згадаємо, що в мові Паскаль ; після оператора можна було й не ставити, якщо далі йде роздільник (END).

Як відомо, у Паскалі оператори розміщалися за розділом оголошень. Тут же описи можуть бути й до початкової дужки, і після й ще далі.

Складений оператор - це послідовність операторів, обмежених фігурними дужками. Іншими словами - це блоки. Розміщення операторів у рядку - довільне. Будемо розміщати дужки, які відкриваються й закриваються, на одній вертикалі, а внутрішні оператори - трошки праворуч. Одержимо текст із відступами, який краще сприймається.



ПРИКЛАД ПРОГРАМИ

```
/* арабське число пишеться римськими літерами */
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
//розділ опису глобальних змінних та функцій
long roman(long,int,char); //заголовок функції
void main(void)
{ long num;
  int rnum[7]={1000,500,100,50,10,5,1};
  char symb[7]={'M','D','C','L','X','V','I'};
  do
  {clrscr(); //очищення екрана
   printf("1000-M 500-D 100-C 50-L 10-X 5-X 1-
I\n");
   print(" Введіть натуральне число (арабське)->
");
   if(!(scanf("%ld",&num))||(num<0)) //
перевірка правильності введення
   { printf("\n Помилка вводу ");
     printf("\n Для продовження натисніть будь-
яку клавішу ");
     getch();
     num=0;
     fflush(stdin); // очищення буфера введення
   }
  } while (!num);
  printf(" Римське позначення числа -> ");
  // цикл визначення римских позначень
  for (int i=0;i<=6;i++)
  {   num=roman(num,rnum[i],rsymb[i]);  }
  getch();
} // завершення основної функції
// опис функції
long roman(long n1,int n2,char symb)
{
 while (n1>=n2)
 {
  putch(symb); // виводить символ на екран
  n1-=n2;
 }
 return (n1);
}
```

1. Директива **#include**

Директива #include включає в текст програми зміст вказаного файла. Ця директива має дві форми:

```
#include "имя файла"
#include <имя файла>
```

2. Директива **#define**

Директива #define служить для заміни констант, що часто використовуються, ключових слів, операторів або виражень деякими ідентифікаторами.

Має дві синтаксичні форми:

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

Приклад:

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
#define  MAX(x,y) ((x)>(y))?(x):(y) //      t=MAX(i,s[i])  → t=((i)>(s[i])?(i):(s[i]);
```

3. Директива **#undef**

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

Приклад:

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

Ці директиви скасовують визначення іменованої константи WIDTH і макровизначення MAX.

Лекція 4

Розділ 2. Класи пам'яті. Логічні вирази. Керуючі структури.

2.1 Класи пам'яті

Кожна змінна крім типу визначається своєю областю дії й часом існування (протягом виконання всієї програми, або лише в окремому блоці). Для підвищення ефективності використання пам'яті програма складається з 4 частин (сегментів):
1. Сегмента *коду*, де розміщаються ***коди програм***;
2. Сегмента *даних*, де розміщаються глобальні й статичні ***змінн***і, які існують протягом усього часу виконання програми;
3. Сегмента *стека*, де розміщаються локальні й реєстрові ***змінні***;
4. Додаткового *сегмента*.
Крім цього, є ще п'ята область - для запису ***динамічних змінних - "куча" (heap).***

Стек - це область пам'яті, у якій запис походить від більших адрес до менших, а зчитування у зворотному напрямку. Інакше говорять, що реалізується дисципліна - перший прийшов, останнім обробився.
Клас пам'яті визначає місце, де розташований об'єкт (внутрішні регістри процесора, сегмент даних, сегмент стека) і одночасно час існування.
У мові С++ існують 4 класу пам'яті:
1. Автоматична (auto);
2. Реєстрова (register);
3. Статична (static);
4. Зовнішня (extern).

ВІДПОВІДНО до класу пам'яті, де розміщається змінна, можна називати й змінну - автоматичної, реєстрової, статичної або зовнішньої.

Автоматичні змінні

Описуються в блоці таким чином

```
{ auto int a=123;
  auto char b;
  auto float c=45.28;
}
```

Зазвичай слово **auto** пропускається. Область дії автоматичних змінних - у межах блоку, після виходу із блоку їх значення стають невизначеними (усі що завгодно). Ініціалізація автоматичних змінних відбувається щораз при вході в блок, тобто в процесі виконання. Тому int **a=123** еквівалентно **int a; a=123**;
Як видне, автоматична змінна є локальної, існує тільки по необхідності й не може бути змінена іншими функціями.
Якщо в деякому блоці втримуються внутрішні блоки, то область дії автоматичних змінних, описаних у зовнішньому блоці, поширюється й на внутрішні. Тобто у внутрішньому блоці їх можна не описувати.

Якщо ж у внутрішньому блоці автоматична змінна вживається в іншому розумінні, то при виконанні дій внутрішнього блоку діють його опис, а не зовнішнього блоку.

```
main()
```

```
/ / Початок зовнішнього блоку
{ int x=1; char z='w';
    if (x=1)
    { // внутрішній блок
          int y=2;
          float z=-345.5674;
          cout<<"y= "<<y<<"z= "<<z<<endl;
     }
  printf (" x=%d zf=%e z=%c\n", x,z,z);
  getch();
}
```

Буде написано: y=2 z= -345.567413
x=1 zf=-7.4204e+41 z=w // ‘м’

Реєстрові змінні

Як відомо, дані можуть розміщатися як в оперативній пам'яті, так і в регістрах. Використання регістрів зменшує кількість пересилань і прискорює виконання програми. Тому можливе використання реєстрових змінних, опис яких має вигляд

```
{ register int x=2;
}
```

Однак оскільки мовам високого рівня регістри недоступні, про їхній стан відомо тільки компіляторові, то використання регістрів буде можливо тільки тоді, коли є вільний регістр, і якщо регістр може вмістити відзначену змінну.
Інакше компілятор розмістить цю змінну в оперативній пам'яті й вона буде звичайної автоматичної змінною. Тому використання реєстрових змінних недоцільне.
Як видне, автоматичні й реєстрові змінні мають локальний характер, тому їх розміщають у сегменті стека.

Статистичні змінні

Задаються описом

```
{ static int a;
  static float b=3.15;
}
```

Як і для автоматичних, область дії статичних змінних є блоком. Відрізняються вони від автоматичних тим, що після виходу із блоку їх значення зберігається. Воно буде початковим при наступному вході в цей блок. Наприклад:

```
void lvars ()
 { int z=0;
   z++;
   printf (“ z=%d\n”,z);
}
main ()
{ lvars();
  lvars();
  lvars();
}
```

другий варіант (static int z=0***)***

```
  void statvars ()
{ static int z=0;
  z++;
```

```
   printf (“ z=%d\n”,z);
}
main()
{ statvars ();
  statvars ();
  statvars ();
}
```

Ці дві функції **lvars()** і **statvars()** відрізняються тим, що в першої z є автоматичної, а в другий - статичній змінній. Це приведе до того, що в першому випадку змінна z щораз буде одержувати значення 0, і тому три рази буде надрукована одиниця.
У другому випадку z є статичною. Вона буде мати значення z=0 лише один раз і тому буде надруковано 1, 2, 3.
Відзначимо, що якщо автоматичної змінної початкові значення надаються щораз при вході до блоку, те статична змінна набуває початкового значення лише один раз - у процесі компіляції. Якщо початкове значення не відзначене, то за **замовчуванням** воно **рівняється нулю.**

Глобальні змінні

Глобальні змінні є глобальними й доступні будь-яким функціям. Вони визначаються поза функціями
Нижче приводиться дуже невеликий приклад закінченої програми, у якій рядок описується в одному файлі, а її печатка проводиться в іншому. Файл vars.h визначає необхідні типи

```
// vars.h
extern char* sms;
extern void global();
```

В файлі main.cpp знаходиться головна програма:

```
// main.cpp
#include “vars.h”
#include “gl.h”
char *sms = “Тест глобальних змінних”;
int main()
{ global(); }
а файл gl.h друкує рядок:

// gl.h
#include <iostream.h>
#include “ vars.h”
void global( )
{  cout << sms << “\n”;  }
```

Вивід на екран: **Тест глобальних змінних**

Однак, якщо опис глобальних змінних передував певній функції, то він може бути опущений. Таким чином, опис глобальних змінних діє на всі функції, які розташовані нижче. Як і для статичних змінних, ініціалізація зовнішніх змінних відбувається один раз під час компіляції. По умовчуванню привласнюється значення нуль.
Статичні й глобальні змінні розміщаються в сегменті даних. Якщо програми складається з декількох файлів, то дія статичних змінних поширюється лише на один файл, де вони описані. ***Зовнішні змінні діють у межах декількох файлів, але їх опису повинні бути продубльовані.***

У міру можливості слід уникати використання в програмах глобальних змінних. Однак якщо ваша програма повинна використовувати глобальну змінну, то може трапитися, що ім'я глобальної змінної конфліктує з іменем локальної змінної. При виникненні такого конфлікту C++ надає пріоритет локальної змінної. Інакше кажучи, програма припускає, що у випадку конфлікту кожне посилання на ім'я відповідає локальної змінної.
Однак можуть бути ситуації, коли вам необхідно звернутися до глобальної змінної, чиє ім'я конфліктує з іменем локальної змінної. У таких випадках ваші програми можуть використовувати глобальний оператор дозволу З++ (::), якщо ви прагнете використовувати глобальну змінну. Наприклад, припустимо, що у вас є глобальна й локальна змінні з іменем number. Якщо ваша функція прагне використовувати локальну змінну number, вона просто звертається до цієї змінної, як показано нижче:

```
number = 1001; // звернення до локальної змінної
```

З іншого боку, якщо ваша функція прагне звернутися до глобальної змінної, програма використовує глобальний оператор дозволу, як показано нижче:

```
::number = 2002; // Звернення до глобальної змінної
```

Наступна програма. На додаток до цього функція *show_numbers* використовує локальну змінну з іменем *number*. Ця функція використовує оператор глобального дозволу для звертання до глобальної змінної:

```
#include <iostream.h>
int number = 1001; //  Глобальна змінна
void show_numbers(int number)
{
  cout << "Локальна змінна number містить число " << number << endl;
  cout << "Глобальна змінна number містить число " << ::number << endl;
}
void main(void)
{
  int some_value = 2002;
  show_numbers(some_value) ;
}
```

Лекція 5

2.2. Логічні вирази
Відношення, логічні операції, умовні вирази

Існують ***чотири операції відносин***
>, >=, <, <=
і ***дві операції рівності***
= = але !=.

Чотири перші операції між собою рівноправні, операції рівності мають менший пріоритет.

У відносинах арифметичні вираження виконуються раніше, тому у вираженні **i <a+b** додаткових дужок не потрібно. І взагалі на загал в С++ є 11 рівнів пріоритетів операцій, у мові Паскаль лише 4.

У логічних виразах використовуються 3 логічних операції:
1. **!** - "заперечення"
2. **&&** - "І",
3. || - "АБО" з врахуванням старшинства (!, &&, ||).

Особливість виконання логічних операцій полягає в тому, що в мові немає логічних констант і логічних змінних. Тому логічні операції фактично виконуються над ціл показниками, що плавають і. При цьому FALSE (НЕПРАВДА) відповідає арифметичному нулю. Усе, що не є нулем відповідає TRUE (ПРАВДА).

Порозрядні логічні операції

Для роботи з окремими бітами поруч зі звичайними логічними операціями передбачено **6** порозрядних логічних операції:

1. **&** - порозрядне І;
2. | - порозрядне АБО;
3. ^ - порозрядне виключаюче АБО;
4. ~ - доповнення;
5. << - зсув вліво;
6. >> - зсув вправо.

x	y	x&y	x|y	x^y	x~	y~	y<<1	y>>2
0	0	0	0	0	1	1	*	*
0	1	0	1	1	1	0	*	*
1	0	0	1	1	0	1	*	*
1	1	1	1	0	0	0	*	*

Доповнення є унарною операцією, усі інші - бінарні НЕ можуть виконуватися над операціями типу *float* або *double.*
Операції **&**, | можна використовувати для керування окремими розрядами (бітами).
Наприклад:
n=243 (11110011)

c=n&547 (1000100011)
// ***c*** буде присвоєно код результату **1000112=3510**

Потрібно визначити відмінність у виконанні звичайних логічних операцій і порозрядних.

Наприклад:
(1&&2 == 1) (0&&2 == 0) (1&&0 == 0) (1&&3 == 1)
У цьому випадку операція логічного І (&&) виробляє значення 1, якщо обоє операндів мають НЕ нульові значення. Якщо один з операндів рівний 0, то результат також рівний 0. Якщо значення першого операнда рівно 0, то другий операнд не обчислюється.

(1&2 == 0) (0&2 == 0) (1&0 == 0) (1&3 == 1)
У другому випадку перевіряється, є чи хоча б в одній парі бітов одиниці. Оскільки код 1 і код 2 різні, то результат " НЕПРАВДА " або 0.
Операція доповнення ~ у кожному біті змінює 1 на 0 і 0 на 1.
Операція зрушення виконується на таку кількість розрядів, яка відзначена в операнді праворуч **x<<2**. У правих бітах з'являється відповідна кількість нулів. Операція зрушення праворуч **y>>5** виконується в різних машинах по різному:

1. З фіксацією знакового розряду (як арифметичний зсув);

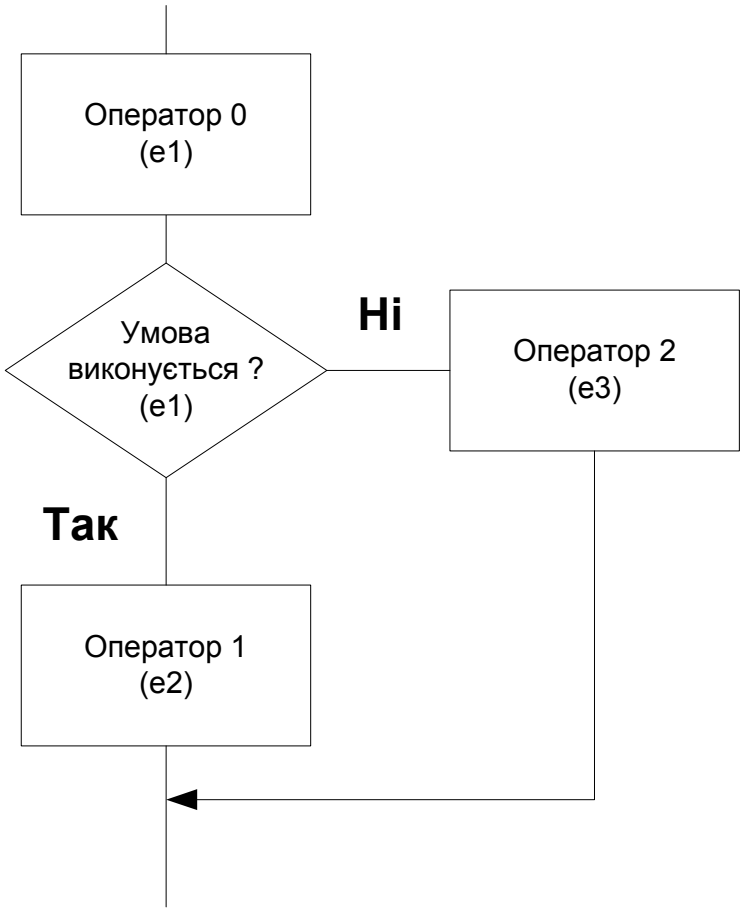
2. Без фіксації (логічний зсув).

Умовні вирази

У деяких випадках умовні оператори можна замінити умовним вираженням, яке має вигляд:

e1 ? e2 : e3,

де **e1**, **e2**, **e3** - вираження. Якщо вираження **e1** одмінне від 0 (ІСТИНА), то виконується вираження **e2**. Інакше - вираження **e3**.



Наприклад, вибір максимального із двох чисел

```
z = a>b ? a : b;        /*z = max (a, b)*/
```

Виконання умовних виразів ефективніше, ніж операторів.

Лекція 6

2.3 Керуючі структури

Визначають, які операнди необхідно виконувати і в якому порядку. Інакше визначають потік управління у програмі. Як зазначалось, Бьом та Якоініні показали, що будь-який алгоритм може бути реалізований за допомогою трьох керуючих структур:

1. послідовне виконання;
2. умовне виконання;
3. цикл.

У мові С будь-який вираз стає оператором, якщо за ним поставити; наприклад:

```
a = 0;   i++; printf ("%d\n", x);
```

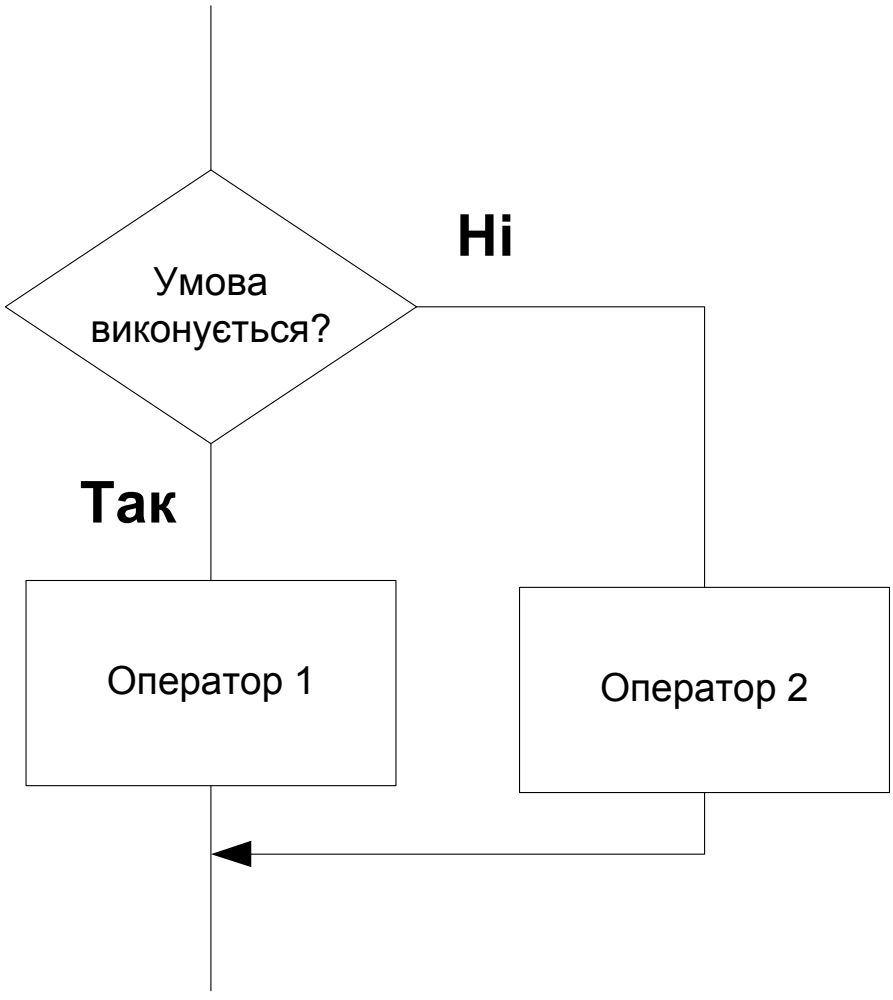
Послідовність операторів, обмежена { } є складеним оператором, або блоком. При цьому після правої дужки ; не вживається.

Умовні оператори

Умовний оператор *if*

Має вигляд:

```
if (вираз) оператор 1;
else оператор2;
```



Приклад:

// перевірка на можливість побудови трикутник

```
main ()
{ float a, b, c;
  int i;
  printf (“ Введіть сторони a, b, c,\n”);
  scanf (“ %f %f %f”, &a, &b, &c);
        i = (a+b>c)&&(b+c>a)&&(a+c>b);
   if (i)
 printf (“ Трикутник можна побудувати \n”);
   else
 printf (“ Трикутник побудувати неможливо !!! \n”);
}
```

Можливе використання й скорочений оператор:

if вираз) оператор1;

Якщо вираз має ненульове значення, то виконується оператор1. Інакше переходимо до виконання наступного за порядком оператора.

Ніякі обмеження на тип операторів не накладають. Тому можна використовувати й вкладені умовні оператори. Приходимо до конструкцій:

if вираз) оператор else if . else if

Оскільки в таких послідовностях можна використовувати як повні умовні оператори, так і скорочені, то виникають неоднозначності:

якщо до оператора віднести **else**

if (n>0) if (a>b) z=0; else z=b; або if (n>0) {if (a>b) z=0;} else z=b;

Діє правило, що саме внутрішнє else ставиться до найближчого ліворуч **if**. Якщо ж потрібно змінити приналежність, то необхідно використовувати фігурні дужки

Перемикач switch

У тих випадках, коли виникає вибір з багатьма можливими результатами, доцільно використовувати перемикач.

Його структура:

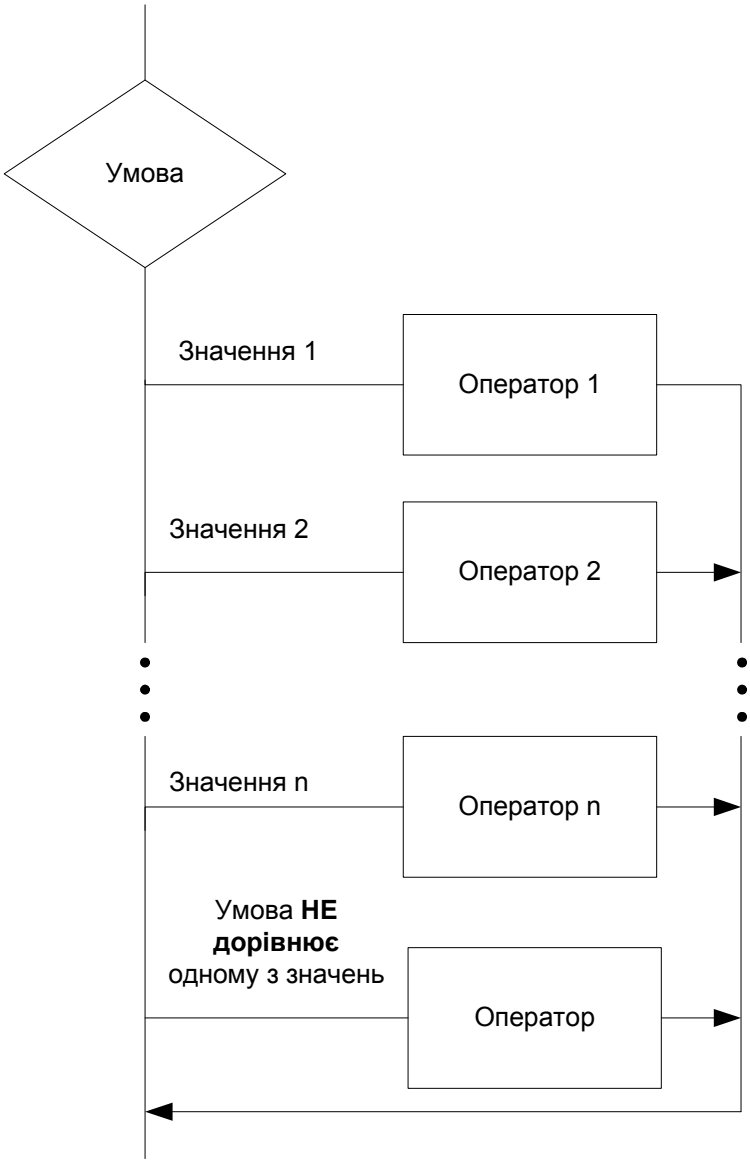
```
switch (цілий вираз)
{case константа 1: оператор 1; [break]
  case константа 2: оператор 2; [break]
……..
case константа n: оператор n; [break]
default :оператор n+1;
}
```

Слово **case** вживається у кожному виборі, в інших випадках, замість констант можна вживати константний вираз.

Оператор виконується в такий спосіб:

1. Виконується вираз в круглих дужках. Його значенням може бути або ціле, або літера (символ);
2. Отримане значення рівняється з константами вибору. Якщо воно збігається з однієї з

константа1 ... константа n, то виконується відповідний оператор i;
3. Далі виконуються всі наступні оператори від i+1 до default (n+1) (відсутність) включно;
4. Якщо значення виразу не збігається з ні однієї константою вибору, то виконується оператор з міткою default.



Наприклад: ввести номер місяця й надрукувати його назва

```
main ()
{ int month;
  printf (“Введіть номер місяця \n”);
  scanf (“ %2d”,&month);
  switch (month);
  { case 1: printf (“ Це січень. \n”);
    case 2: printf (“Це лютий. \n”);
     ……….
    case 12: printf (“Це грудень. \n”);
    default : printf (“Такого місяця немає. \n”);
  }
}
```

Якщо ввести 1, то буде виведено:

```
Це січень.
.....
Це грудень.
Такого місяця немає.
```

Якщо ввести 20, то буде надрукована лише остання пропозиція.
Для того, щоб, вибір був лише один, то в кожний варіант необхідно додати оператор break (вихід, обривши потоку).

```
{ int month;
  printf (“Введіть номер місяця \n”);
  scanf (“ %2d”,&month);
  switch (month);
  { case 1: printf (“ Це січень. \n”); break;
    case 2: printf (“Це лютий. \n”); break;
     ……….
    case 12: printf (“Це грудень. \n”); break;
    default : printf (“Такого місяця немає. \n”);
  }
}
```

Константа може бути і символом:

```
main ()
{ char c;
switch (c)
  { case ‘a’: printf (“Введено символ ‘a’. \n”); break;
    .................
    case ‘z’: printf (“Введено символ ‘z’. \n”); break;
    default : printf (“Такого символу не знаю. \n”); break;
  }
}
```

Ясно, що варіанти можуть розташовуватися в довільному порядку, а не в певному.
Дозволяється кілька констант вибору до одному операторові.

```
switch (month)
{ case 1:
  case 2:
  case 12: printf (“Це зима. \n”); break;
  case 3:
  case 4:
  case 5: printf (“Це весна. \n”);

}
```

Оператори циклу

Цикл с передумовою while

Оператор має структуру:

while (умова) оператор;

Виконується доти , поки умова є істиною. Ясно, що замість одного оператора може бути складений оператор або блок.

Якщо необхідно перервати цикл до його завершення, то можна використовувати оператор break.

Наприклад, визначити, чи є число n простим, тобто ділиться лише саме на себе і на 1:

```
main ()
{ int n, i=1;
  printf (“Введіть ціле число \n”);
  scanf (“ %5d”, &n);
   while (++i<n)
     if (n%i==0)        //остача від ділення
       {printf (“ число НЕ просте \n”); break;}
     if (i==n) printf (“Число просте \n”); }
```

З іншого боку, якщо певну послідовність операторів не треба виконувати та необхідно продовжити цикл (тобто повернутись до його заголовку), то в такому випадку можна використати оператор **continue**.

Наприклад:

для чисел від **1** до **n** надрукувати всі, не кратні 5, тобто ті, які діляться на 5 – не друкувати.

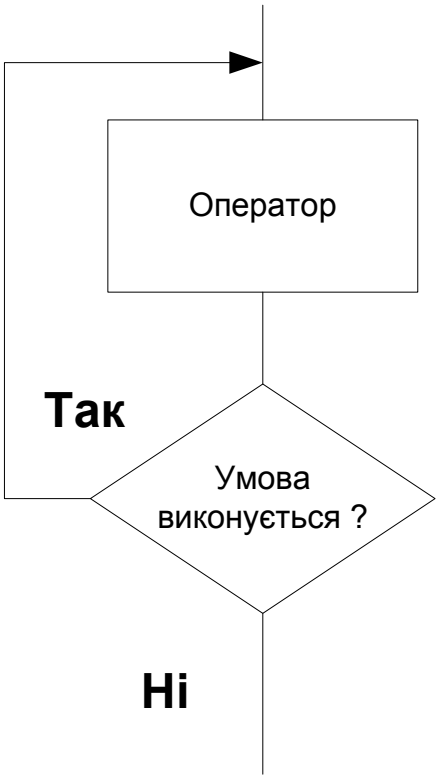
```
main ()
{ int n, i=0;
 printf (“Введіть число \n”);
 scanf (“%5d”,&n);
  while (++i<n)
        { if (i%5==0)
         continue;
         printf (“%5d \n”,i);
         }
```

}

Цикл с післяумовою do-while

Якщо оператори циклу повинні виконуватися принаймні один раз, то можна використовувати інший варіант **do** оператор **while (условие);**

do { оператор} while (умова)



Наприклад: Коректне ведення числа.

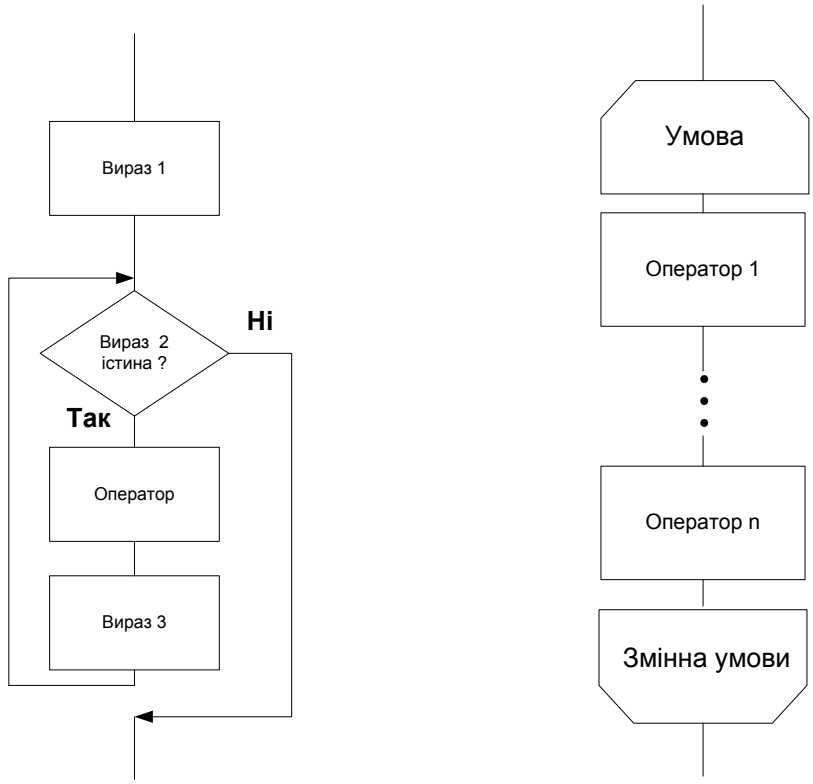
```
main ()
{ int err;
  do{ err=0;
        printf (“Введіть ціле число \n”);
       if (!scanf (“%5d”,&n)) //не більше 5-ти символів
        { printf (Помилка вводу числа \n ”);
          err=1;
        }
  }   while (err);

}
```

Єдина його відмінність в тому, що умова перевіряється після виконання оператора, якщо умова істина, то все повторюється знову.

Оператор цикла for

Має вигляд:

for (вираз 1; вираз 2; вираз 3) оператор;



і виконується таким чином:

вираз 1; while (вираз 2) {оператор; вираз 3;}

Тобто вираз 1 виконується лише один раз. Вираз 2 еквівалентне умові закінчення циклу. Вираз 3 виконується щораз у циклі. Тому: вираз 1 - початкове значення параметра циклу, а вираз 3 - модифікація параметра циклу.

Приклад: Обчислення n!

```
nfactor ()
{ int i, n, nfact=1;
   printf (“Введіть ціле число \n”);
  scanf (“%2d”,n);
    for (i=1; i<=n; i++)          nfact*=i;
    printf (“ Факторіал %2d =%6d \n”, n, nfact);
}
```

На відміну від інших мов програмування лічильник циклу - це просто звичайна змінна. Тому можлива модифікація лічильнику в самому циклі. Після виходу із циклу значення лічильника запам'ятовується.

У загальному випадку, вирази 1-3 мають більш широкий зміст.

Наприклад, вони взагалі на можуть бути відсутніми, але два знаки "**;**" повинні зберігатися обов'язково:

```
for (i=1;;i++) printf(“Нескінченний цикл”);
```

Вийти з нескінченного циклу можна за допомогою оператора **break**.
Замість виразів 1-3 можна використовувати звертання до функцій:

```
for (putchar (‘a’); putchar (‘b’); putchar (‘c’))

putchar (‘d’);
```

Маємо нескінченний вивід abdcbdcbdcbdc...bdc.
Допускається використання й вкладених циклів.

При цьому можна одержати дуже компактні конструкції, зрозуміти які буває дуже складно.

Наприклад: Швидке сортування Шела (1959).

```
void main(void)
{ int v[]={5,-4,0,-17,68,36,90,-7,5,4,12,1,9};
  int gap, i, j, k,temp,n=13;
   for (gap=n/2; gap>0; gap/=2)
    for (i=gap; i<n; i++)
     for (j=i-gap; j>=0&&v[j]>v[j+gap]; j-=gap)
     {
       temp=v[j];
       v[j]=v[j+gap];
       v[j+gap]=temp;
     }
  for (k=0;k<n;k++) cout<<"  "<<v[k];
  getch();
}
```

Масив розбивається на групи, кожна з яких складається із двох елементів. Відстань між парами елементів d=n/2, де n - кількість елементів масиву.Елементи- Пари рівняються між собою і якщо потрібно, то міняються місцями. Потім групи попарно зливаються. Кожна нова група має 4 елемента, відстань між елементами d=d/2. У середині групи виконується сортування, потім групи зливаються. Процес триває доти , поки відстань між елементами не стане 1. На цьому етапі масив сортується методом вставки (пухирця).

Конструкції gap/=2 і j/=gap значить відповідно gap=gap/2 і j=j-gap. Зовнішній цикл змінює зрушення між парою елементів, які рівняються, від n/2, n/4 до 1. Наступний внутрішній цикл забезпечує перегляд масиву, починаючи з n/2, n/4 і 1- го до кінця.

Третій цикл (по j) до деякої міри є фіктивним: у ньому для певної пари для випадку, коли перший елемент більше другого, вони переставляються. Після цього j-=gap, тобто стає менше нуля й відбувається вихід із циклу.

Іноді в циклі for можна відразу використовувати два індекси. У цьому випадку одне вираз поєднує два за допомогою оператора ",": вираз 1, вираз 2. Кома розділяє два вирази, які виконуються зліва направо. Значення й тип операнду визначається правим виразом. Лівий вираз є ніби другорядним.

Наприклад: Переставити символи рядка

```
reverse (char s[])
{ int c, i, j;
printf (“Ввести рядок \n”);
scanf (“ %s”,s);
  for (i=0, j=strlen (s)-1; i<j; i++, j--)
```

```
  { c=s[i];   s[i]=s[j];
    s[j]=c;
  }
}
```

Будемо використовувати два індекси: i- символ спочатку рядка, а j- й кінця, i буде збільшуватися, а j - зменшуватися. Обмін символами продовжуємо поки i<j. Функція strlen - визначає кількість букв у рядку.

```
5  -4  0  -17  68  36  90  -7  5  4  12  1  9
(5,90)(-4,-7)(0,5)(-17,4)(68,12)(36,1)

 5  -7  0  -17  68  36  90  -4  5  4  12  1  9

 5  -7  0  -17  12  36  90  -4  5  4  68  1  9

 5  -7  0  -17  12  1  90  -4  5  4  68  36  9
 5  -7  0  -17  12  1  9  -4  5  4  68  36  90
  -17  -7  0  5  12  1  9  -4  5  4  68  36  90
 -17  -7  0  5  -4  1  9  12  5  4  68  36  90
 -17  -7 0  5  -4  1  4  12  5  9  68  36  90
 -17  -7  0  4  -4  1  5  12  5  9  68  36  90
  -17  -7  0  -4  4  1  5  12  5  9  68  36  90
 -17  -7  -4  0  4  1  5  12  5  9  68  36  90
 -17  -7 -4  0  1  4  5  12  5  9  68  36  90
 -17  -7  -4  0  1  4  5  5  12  9  68  36  90
 -17  -7  -4  0 1  4  5  5  9  12  68  36  90
 -17  -7  -4  0  1  4 5  5  9  12  36  68  90
```

Лекція 7

Розділ 3. Масиви та показчики
3.1 Масиви

Масив - це послідовність однорідних даних, яка має фіксовану довжину. Цей тип належить до похідних, тому що складається із простих.
Масиви, як і інші змінні, повинні бути описаними. Спеціальних ключових слів ні, тобто відзначається ім'я в прямокутних дужках кількість елементів:

```
int a[10]
float x[100].
```

Такий опис може включатися у звичайні змінні char c, d, f[20], де c, d - звичайні символьні змінні. Кожний елемент масиву визначається своїм номером a[2], x[50].

На відміну, від інших мов, нумерація елементів починається з нуля й закінчується номером N-1, де N - загальна кількість елементів. Тому a[2] - це третій елемент, а x[50] - п'ятдесят перший. Це зроблене тому, що повне ім'я масиву є базовою адресою, яка рівняється адресі першого елемента. Тобто a==&a[0] Адреса другого елемента визначається як базова адреса плюс один &a[1]==a+1 Адреса N- го елемента рівняється базової плюс N-1. Ясно, що зсув по адресах фізичних змінних залежить від типу даних, тобто індекс необхідно помножити на відповідний коефіцієнт типу (для char це буде 1, для int 2, float 4 і т.д.). Тому, така система нумерації спрощує адресацію елементів масиву.

У деяких випадках можна не відзначати кількість елементів масиву, наприклад int arr[]. Відповідний розмір визначає сам компілятор. Як і інші змінні масиви зв'язуються з відповідним класом пам'яті. Як і інші змінні масиви пов’язуються з відповідним класом пам'яті. Тому якщо масив відноситься до автоматичного, то його необхідно описувати в блоці. Коли масив є зовнішній і був описаний раніше, то в блоці його можна не описувати. Наприклад:

```
int arr[20];
.....
addum (arr, size)
int arr[], size;
```

У початковій версії мови ініціалізувати можна було лише зовнішні й статичні масиви

```
Static int numb[5]={1, 2, 3, 4, 5};
```

Для автоматичних масивів цього робити не можна було. Версія ANSI-C зняла ці обмеження.

Приклад: обчислення скалярного добутку

```
// скалярний добуток
main ()
{ float x[20], y[20], scal=0;
  int i, size;
  printf (“Введіть розмір масиву =”);
  scanf (“ %2d”, &size);
  printf (“Введіть елементи масиву x \n”);
        for (i=0; i<size; ++i)  scanf (“ %f”,&y[i]); // введення елементів масиву
        for (i=0; i<size; ++i)  scal+=x[i]*y[i]; // добуток
        printf (“ scal=%f \n”, scal); // вивід на екран результату
}
```

У цій програмі індекс задається цілою змінною i, але в загальному випадку він може визначатись цілим виразом, до якого входять цілі змінні та цілі константи.

Багатовимірні масиви

Масив може складатись не лише із даних простих типів, але і з похідних. Зокрема можна розглядати масив масивів. Приходимо до багатомірних масивів 2x, 3x, і т.д. Двомірний масив – це одномірний масив, елементами якого є одномірні масиви. Тому при описі двомірного масиву розмір по кожному виміру зазначається в окремих дужках int a[10][10].
Окремий елемент a[2][3]. Перший індекс – рядка, другий стовпчика. Тому це буде елемент з третього рядка та четвертого стовпчика.

float matr[m][n]

00	01	02	…	***0n***
10	11	12	…	1***n***
…	…	…	…	…
m0	***m1***	***m2***	…	***mn***

Порядок розташування багатомірних масивів у пам'яті такий, що першим змінюється самий правий індекс. Отже, для 2- мірного масиву звичайне розташування - рядками.

A[2][2] a00 a01 a10 a11

Вище відзначалося, що іноді в деяких випадках можна не наводити розміри одномірних масивів. Але для багатомірних це не так. І насправді , якщо у функції записати int array[][], те буде незрозумілим, як такий масив ділити на рядки. Тому обов'язково потрібно відзначати кількість стовпчиків int array[][4], тобто в кожному рядку по 4 елемента. Початкові значення багатомірних масивів задаються в такий спосіб:

```
int z[3][2]={{1, 2},{4, 5},{7, 8}};
```

Оператор goto

Обробка елементів багатомірних масивів часто здійснюється за допомогою вкладених циклів. При цьому для завчасного припинення циклу по певних умовах можна використовувати оператор ***break***. Для того, щоб обійти деякі оператори не виходячи із циклу, застосовують оператор ***continue***.

Але оператор ***break*** здійснює вихід тільки з одного внутрішнього циклу. Тому при значній кількості вкладених циклів для повного припинення циклів краще використовувати оператор безумовного переходу.

З погляду структурного програмування слід уникати цього оператора. Але саме в такому випадку він є найбільше доцільніше.

Нагадуємо, що мітка в мові С - це звичайне ім'я, а не ціле без знака. Мітка спеціально не описується:

Приклад: у матриці розміром 20x20 цілих чисел знайти перше від’ємне число та надрукувати його координати.

```
main ()
{ int a[20][20], i, j, size1, size2;
  // визначення розмірів та значень елементів масиву
      for (i=0; i<size1; i++)
            for(j=0; j<size2; j++)
```

```
                    if (a[i][j]<0) goto label;
  label: printf (“ від‘ємне число %4d має координати [%2d] [%2d] \n”, i, j);
}
```

Строкові масиви

Рядок – це одномірний масив символів, який закінчується нульовим символом. Як зазначалось, рядок береться у дужки “Скоро весна!”.

Її можна присвоювати певній змінній, якщо остання описана як символьний масив char string[]=”Скоро весна!”.

Такий масив можна вводити та виводити з використанням символу перетворення **s**. При цьому вивід триває до появи нуля.

Ніяких дій над такими масивами не передбачено. Для цього використовують стандартні функції мови (бібліотека string.h):

1. Приєднання рядка (конкатенація) char str1[], str2[];

strcat (str1, str2)

До рядка str1 приєднується рядок str2. Видаляється Ø після str1. Результат присвоюється str1. При цьому str2 не змінюється.

2. Порівняння двох рядків

strcmp (str1, str2)

strcmp (str1, str2)=str1-str2

strcmpi (str1, str2) та

stricmp (str1, str2) – вважає малі й великі букви однаковими Значення цієї функції менше Ø, якщо лексикографічно str1 раніше str2, дорівнює Ø, якщо вони співпадають та більше Ø – в іншому разі.

3. Копіювання рядка

strcpy (str1, str2)

strcpy (str1, str2)

рядок str2 копіюється в str1, str2 не змінюється.

4. Визначення кількості символів у рядку без обліку заключного нуля

strlen (str1)

strlen ("abcde") - 5.

5. Перетворення рядка str в число подвійної точності

double atof(char *str)

```
char str3[ ]="-345.575784876";
printf("\n\n6. str3=%s, atof(str1) %20.18f",str3,atof(str3));
strcpy (str3, "-345.54563782rt75");
printf("\n\n6. str3=%s, atof(str1) %20.18f",str3,atof(str3));
```

```
str3=-345.575784876, atof(str1) -345.575784876000000000

str3=-345.54563782rt75, atof(str1) -345.545637820000024000
```

- Розпізнає символьна вистава числа із плаваючою крапкою, якщо символи відповідають формату: [пробіли] [знак] [ddd] [.] [ddd] [e|e[знак]ddd], де [ddd] - числа;[e|e] - покажчик показника ступені;
- Припиняє перетворення на першому неопізнаному символі. У випадку переповнення ***atof*** повертає +(-)HUGE_VAL, глобальна змінна errno установлюється в ERANGE.

6. Перетворення рядка str у число подвійної точності без втрати значення (бібліотека <stdlib.h>):

***strtod*(*str1,&str2*)**

```
char string[20], *stopstring;
 float x;
 string="3.1415926stop";
 x=strtod (string, &stopstring);
 printf (" x=%.7f  - %s\n", x, stopstring);
```

На экрані: x=3.1415925 – stop

7. Перетворення рядка str у десяткове ціле (бібліотека stdlib.h)

atoi (str1)

- Розпізнає символьне представлення десяткового числа, якщо символи відповідають формату: [пробіли] [знак] [ddd], де [ddd] - числа;
- Припиняє перетворення на першому не розпізнаному символі;
- У випадку переповнення atoi повертає +(-)HUGE_VAL, глобальна змінна errno установлюється в ERANGE.

8. Перетворення рядка str у десяткове довге ціле

strtol (** const char **str1,** char ***error,** int **base)

Перетворює рядки ***str1*** в довге ціле відповідності з указаним **base**, яке повинне знаходитися в діапазоні: 2- 36 включно або бути рівним нулю.

9. Перетворення цілого числа в рядок

itoa (int v, char *str, int baz)

Функція перетворить символи числа v у символьний рядок, що закінчується Null- символом, і запам'ятовує результат в str. Аргумент baz визначає підстава системи числення для v; його значення може лежати в межах від 2 до 36. Якщо baz = 10 і v - негативне число, то першим символом у рядку результату буде знак мінус.

Самостійно переглянути:

ltoa
utoa
ecvt
fcvt
gcvt

Лекція 8

3.2 Покажчики

Для всіх відзначених раніше типів даних ім'я - це символічне позначення місця в пам'яті. Як відомо, після компіляції вихідної програми ці імена заміщаються відповідними адресами. Під час виконання програми використовуються значення, які зберігаються по певних адресах. Наприклад, в операторі присвоєння var1=var2; значення змінної var2 копіюється в змінну var1.

Поруч із такими змінними в мові C++ передбачаються й такі, які зберігають адреси змінних або покажчики.

Покажчик - це змінна, що містить адреса інший змінної.

Відомо, що покажчики застосовуються й у мові Паскаль:

```
TYPE  POINT=^REAL;
```

Припустимо, що x - змінна, наприклад, типу INT, а px - покажчик, створений якимось ще не зазначеним способом.

Унарна операція & передає адресу об'єкта, так що оператор px=&x. присвоює адресу x змінної px; говорять, що px "вказує" на x. Операція & застосовна тільки до змінних і елементів масиву, конструкції виду (&x-1) і &3 є некоректними. Не можна також одержати адресу реєстрової змінної.

```
int* p, y;       // int* p; int y; НЕ int* y;
int x, *p;       // int x; int* p;
int v[10], *p;   // int v[10]; int* p;
```

Операції над покажчиками

Якщо визначені змінні ***x, i***

```
int x, i;
```

то поруч зі звичайними діями можна оперувати й з їхніми адресами:

```
px=&x; //& взяти адресу
```

за допомогою операції **&** (взяти адресу). Відзначимо, що праворуч від **&** може стояти тільки ім'я змінної або змінної з індексом, а не вираження загального виду.

Унарна операція * розглядає свій операнд як адресу кінцевої мети й звертається по цій адресі, щоб витягти вміст.

Змінні, що брати участь у всім цьому необхідно описати:

```
int x=10, y,*px; // x=10
px = &x;         // px – адреса змінної x
y = *px;         // y=x
```

Відзначимо, що в описі покажчика обов'язково визначається тип об'єкта, з яким визначений покажчик. Цей опис використовується при обчисленні вираїв із покажчиками.

```
char c1 = 'a';
char* p = &c1;   // в p зберігається адреса c1
char c2 = *p;    // c2 = 'a'
```

Покажчики можна використовувати в операторах присвоєння px=py, якщо px і py відповідають однаковим типам даних.

Єдиною константою, яку можна присвоїти покажчику, є нуль, або NULL, що у файлі stdio.h : py=NULL.

До покажчиків можна додавати або віднімати ціле число:

py=px+2; pz=py-4 - виходить, що покажчик px необхідно збільшити на дві одиниці даних. Ясно, що коли px і py є покажчиками символів, те це буде два байти, або просто два. Коли ж px і py є покажчиками дійсних чисел з подвійною точністю double *px, *py; то це буде 16 байт або 16. тому, залежно від типу даних у вираженнях з покажчиками здійснюється відповідне масштабування.

Для покажчиків визначена й операція розрахунку різниці адресів: pz=px-py.

Операція додавання покажчиків немає ніякого змісту. Оскільки визначені операції додавання й вирахування констант, те визначені й операції збільшення й зменшення

px++; py--; px+=i.

Крім того **==, !, =, >, <.**

В арифметичних виразах замість імені змінної можна застосовувати операцію доступу за покажчиком. Коли:

```
int x,y,*px;
if (px==&x) y=*px+1; //y=x+1
else printf (“%d \n”, *px);.
```

Операції доступу за покажчиком можна вживати й у лівій частині оператора присвоєння:

```
px=0;
 x=0;
```

або *px+=1;

x=x+1;

Унарні операції * і **&** виконуються раніше арифметичних. Тому:

```
y=*px+1;
y=x+1;
y=*(px+1); // привласнює змінній y значення комірки з покажчиком px+1
```

Між собою унарні операції рівноправні й виконуються праворуч ліворуч

*px++ // **y** значення комірки з покажчиком px+1

і

(*px)++ //x+1

також ++*px.

Для покажчиків можна застосовувати операцію взяти адресу, де перебуває сам покажчик &px.

От, наприклад, функція, що підраховує число символів у рядку (не враховуючи завершального 0):

```
int strlen(char* p)
{
   int i = 0;
   while (*p++) i++;
   return i;
}
```

Інший спосіб знайти довжину полягає в тому, щоб спочатку знайти кінець рядка, а потім відняти адрес початку рядка від адреси її кінця:

```
int strlen(char* p)
{  char* q = p;
   while (*q++) ;
   return q-p-1;
}
```

Наприклад, використання постфіксной операції збільшення робить наступні цикли **while** ідентичними:

```
while (*string) {   cout << *string++;}
while (*string) {   cout << *string;   string++;}
```

Крім того, визначені відносини:

```
==   !   =   >   <
```

В арифметичних виразах замість імені змінної можна застосовувати операцію доступу за покажчиком.
Коли:

```
int x,y,*px;
if (px==&x) y=*px+1; // y=x+1
else printf ("%d \n", *px);
```

Лекція 9

3.3 Покажчики і масиви

Звичайно покажчики рідко вживаються для простих даних. Частіше вони зв'язуються зі складними типами, зокрема, з масивами.

У мовах високого рівня, наприклад Pascal, операції з елементами масивів реалізуються індексною арифметикою. У машинно-орієнтованих мовах ці операції реалізуються непрямою адресацією, яка є швидшою. Ця можливість існує й у мові С++ і полягає у використанні покажчиків. Тому покажчики й масиви тісно зв'язані.

Приклад: посимвольно надрукувати рядок " Тестування !" ***у прямому порядку***

```
void main(void)
{ char *str=" Тестування !";
  while(*str) putchar(*str++);
}
```

Їхнє використання ефективніше, чим з індексною арифметикою. Але зрозуміти такі програми важче.

Можна зробити деякі порівняння. Повне ім'я масиву є адресою базового елемента, покажчик також набуває значення адреси. Тому іноді говорять, що повне ім'я масиву є покажчиком. Але це не можна розуміти буквально.

У чому ж полягає різниця між ними?

Якщо задати рядок масивом char st[]="Тестування !", під час трансляції під масив буде виділена постійна ділянка пам'яті

st: “ Тестування!” ‘0’,

де повністю розміщений рядок разом із заключним нулем.

Ім'я масиву рівняється базовій адресі цієї ділянки. Тобто на увесь час виконання програми за ділянкою пам'яті закріплюється ім'я масиву. Тому воно є константою і його не можна змінювати під час виконання програми за допомогою оператора присвоєння. Наприклад st="~~spring~~"; або st=pr, де pr деякий покажчик на символ. Такі спроби розцінюються як спроба змінити адресу, тобто константу, і кваліфікуються як помилка при виконанні. От чому в мові С++ не можна вживати операції присвоєння для масивів.

Зміст масивів можна змінювати, наприклад, за допомогою функції копіювання або введення.

Якщо задати рядок покажчиком

char *pr=”Тест!”; виходить, що під час трансляції рядковій константі десь виділена пам'ять і початкова адреса цього рядка записаний у покажчику

pr

pr “Тест!”

Покажчик pr не є константою, а змінної. Тому його значення в процесі виконання можна змінювати *pr="spring";

Цей оператор не означає, що замість рядка "тест!" на його місце записується "spring". Просто в пам'яті виділяється нове місце під рядок "spring" і його базова адреса записується в покажчику pr

“Good st!”

pr

“Spring”

Спроба змінити саму константу "good st!" *pr=:Spring; розглядається як помилка. Потрібно відзначити, що при роботі з масивами за допомогою покажчиків розмір масивів не контролюється й такі можливості не фіксуються виконуючою системою.

Якщо покажчик не буде ініціалізуватися, то його значення невизначене (кожне). Тому можливі помилки при використанні покажчиків.

Наприклад:

```
main ()
{ int *a;
  *a=25;
  printf (“*a=%d\n”, *a);
}
```

У таких випадках необхідно спочатку виділити певна ділянка динамічної пам'яті (під час виконання). Це можна зробити за допомогою функції malloc:

(тип *) malloc (кіл-ть байт);

```
char *ptr;
ptr=(char*) malloc (100); // виділяється 100 байт, початкова адреса записується в ptr
```

Функція **malloc** повертає **char**- покажчик на закріплений простір.

У пам'яті, на яку вказує значення, що вертається, гарантоване вирівнювання для зберігання будь-якого типу об'єкта. Щоб одержати покажчик на тип, відмінний від char, використовується перетворювач типу значення, що вертається.

Вертається значення NULL, якщо вільної пам'яті залишилося мало.

```
#include <malloc.h>
int *intarray;
// виділяється місце для 20 цілих значень
intarray=(int*)malloc(20*sizeof(int));
```

Для звільнення динамічної пам'яті використовується функція:

free (pr),

де **pr** - ім'я покажчика.

Тому попередній фрагмент правильно треба було б написати так:

```
main ()
{ int *a;
  a=(int *) malloc (sizeof(int));
  *a=25;
  printf (“*a=%d\n” *a);
  free(a);}
```

Ще одна функція виділення певної ділянка динамічної пам'яті (під час виконання)

(тип *) calloc (кількість елементів n, розмір елемента в байтах size);

Функція calloc виділяє простір для зберігання масиву з n елементів, кожний довжиною size байт. Кожний елемент ініціалізується в 0. Повертає значення NULL, якщо залишилося недостатньо пам'яті.

free(ім'я покажчика) - звільняє захоплене місце.

Ще одна функція виділення певної ділянка динамічної пам'яті (під час виконання) C++.

ім'я покажчика = new тип[кіл-ть];- виділяє простір для зберігання масиву з [кількість] елементів

delete ім'я покажчика - звільняє виділене місце.

Функція ***realloc*** змінює розмір раніше захопленого блоку пам'яті. Вміст блоку не змінюється.

(тип *) realloc (новий розмір у байтах);

Функція **realloc** повертає покажчик на перезахоплений блок пам'яті.

```
#include <malloc.h>
#include <stdio.h>
char *t1;
t1=(char*) malloc(50*sizeof(char));
/ / Перевиділяє блок, який містить 100 символів
if (t1 != NULL)
  t1=realloc(t1,100*sizeof(char));
```

Покажчики і багатовимірні масиви

Існує двувимірний масив:

```
int matr[3][2];
int *pr;
```

Ясно, що им’я matr дорівнює адресу базового елемента matr==&matr[0][0]; тому після присвоєння pr=matr

pr+1==matr[0][1]
pr+2==matr[1][0]
pr+3==matr[1][1]
і т.д.

Отже за допомогою такого покажчика й відповідного зсуву можна адресувати будь-який елемент такого масиву.

Як відомо, елементи двомірного масиву розташовуються в пам'яті рядками:

matr[0]	matr[0][0]	matr[0][1]
matr[1]	matr[1][0]	matr[1][1]
matr[2]	matr[2][0]	matr[2][1]

Двомірний масив - це є масив, який складається з масивів, і повне ім'я масиву є адресою базового елемента. Тому перший рядок буде мати ім'я matr[0], другий matr[1], третій matr[2]. Ці імена мають значення адрес перших елементів відповідного рядка.

matr[0]=&matr[0][0]=matr matr[1]=&matr[1][0] и т.д.

Із цього випливає, що поруч із повним іменем масиву matr, іменами окремих елементів matr[i][j] можна вживати й імена окремих рядків matr[i]. (В інших мовах, наприклад, Паскалі коли масив двомірний, то вживання імені з одним індексом є помилкою).

Отже, відкривається можливість працювати із двомірним масивом через одномірний. Із цього випливає також, що двомірний масив може бути заданий як одномірний масив покажчиків.

При цьому кількість елементів у рядку не фіксується. Це зручно при роботі з масивами рядків. Наприклад, написати функцію, яка за номером місяця повертає його назву:

{char * name [] = {"Немає такого місяця", "Січень", "Лютий", "Березень", "Квітень", "Травень", "Червень", "Липень", "Серпень", "Вересень", "Жовтень", "Листопад", "Грудень"};
return (n <1 | | n> 12? name [0]: name [n]);

Тут кількість елементів у масиві покажчиків name визначається кількістю рядків у фігурних дужках.

Для того, щоб скопіювати рядок, зовсім не потрібно його фізично копіювати, а досить лише дорівняти відповідні покажчики. Також при сортуванні можна переставляти не рядки, а їх покажчики.

Розглянемо програму введення, сортування й виведення рядків. Використовуємо два масиви: рядків char **str і масив покажчиків char *tstr, який зв'яжемо з масивом str, де nums - максимальна кількість рядків, а numc - розмір рядків;

Сортування будемо проводити перестановкою покажчиків у масиві str. Сам же масив str буде залишатися незмінним.

Для порівняння рядків будемо використовувати функцію **strcmp (str1, str2)**, аргументами якої є імена рядків або відповідні покажчики.

Для сортування використаємо метод бульбашки із змінним зсувом, який розглядали раніше.

```
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>
char **str,*tstr;
void main(void)
{ int a,b,nums,numc;
  clrscr();
  cout<<" Кількість рядків = "; cin>>nums;
  cout<<" Кількість символів = ";cin>>numc;
// виділення за допомогою new
/*  str= new char*[nums];
     for(a=0;a<nums;a++) str[a]= new char[numc+1];
*/

// виділення за допомогою calloc
str=(char**)calloc(nums,sizeof(char*)); виділення пам'яті для масиву рядків
  for(a=0;a<nums;a++) str[a]=(char*)calloc(numc+1,sizeof(char));// для самих рядків
  randomize();
  for(a=0;a<nums;a++){
   b=0;
   do{
    int ch=random(257);
    if ((ch>64 && ch<91) || (ch>96 && ch<123)) {
     str[a][b]= ch;
     b++;
    }
   }while (b<numc);
   str[a][b]= '\0';
  }
  cout<<"\nСгенерований масив з ["<<nums<<"] рядків ";
  for (a=0;a<nums;++a) cout<<"\n["<<a+1<<"] рядок -> "<<str[a];
//сортування по зростанню
   for(a=1;a<nums;++a)
   for(b=nums-1;b>=a;--b){
```

```
    if(strcmp(str[b-1],str[b])>0){
      tstr=str[b-1];
      str[b-1]=str[b];
      str[b]=tstr;
    }
   }
//
   getch();
   cout<<"\n\nВідсортований масив з ["<<nums<<"] рядків ";
   for (a=0;a<nums;a++) cout<<"\n ["<<a+1<<"] рядок -> "<<str[a];
// звільнення пам‘яті
   for (a=0;a<nums;a++) free(str[a]);
   free(str)
   getch();
}
```

Лекція 10

Розділ 4. Функції. Структури.

4.1 Функції та їх використання

Як відзначалося, програма мовою С++ складається з декількох функцій. Усі вони рівноправні, але обов'язково якась одна повинна мати ім'я main і компіляція починається саме із цієї функції. Для більшої ясності програму краще виконувати з невеликих функцій, а не з більших.

Імена функцій мають глобальний характер, тому вкладених функцій не передбачене. Послідовність обігу – будь-яка, може викликати функцію із цієї ж самої функції, тобто рекурсивні функції.

Як зазначалося, програма мовою С складається з декількох функцій. Всі вони рівноправні, але обов’язково якась одна повинна мати ім'я **main** і компіляція починається саме з цієї функції. Для більшої ясності код програми краще виконувати з невеличких функцій, а не з великих.

Імена функцій мають глобальний характер, тому вкладених функцій не передбачено. Послідовність звертання – будь-яка, дозволяється викликати функцію з цієї ж самої функції, тобто рекурсивні функції.

Структура функції

У загальному випадку програма містить кілька функцій. Тому необхідно визначити структуру функції - підпрограми.

Опис функції являє собою блок, тобто складається із заголовка й тіла.

Заголовок має вигляд: ¶

Тип_функції ім‘я_функції (тип змінна, тип змінна, ...)

{

Тіло функції;

return (вираз);

}

де ***тип змінна*** – список формальних параметрів.

На відміну від звертання до функції при описі функції наприкінці ";" не ставиться. З іменем функції можна зв'язувати одне або жодного значення. Наприклад, функція **puts** виводить тільки рядок і з її іменем не зв'язується ніяке значення.

Щоб ім'я функції одержало значення (повертало значення), у її тілі повинні бути присутній оператор (функція) **return (вираз)**. Дужки не обов'язкові.

Значення вираження привласнюється імені функції, тобто функція повертає значення змінній. Інше призначення цього оператора - передати керування відповідній до програми на оператор, який іде слідом за оператором виклику функції.

Функція може містити один, декілька операторів (функцій) **return** або жодного.

Приклад: використання функції визначення абсолютної величини:

```
main ()
{ int a=15, b=0, c=-32;
 int d, e, f;
 d=absn (a);
 e=absn (b);
 f=absn (c);
 printf (“ %d %d %d \n”, d, e, f);
}
/*функція абсолютного значення*/
```

```
absn (int x)
{   return (x<0 ? -x : x);}
```

Тут значення змінної **y** буде привласнено імені функції **absn**, яке передає це значення в визивну функцію. Відзначимо, що змінна **y** є локальною (автоматичною) і без неї можна обійтися.

Фактичні і формальні параметри

Функція уявляє собою певну закінчену процедуру і її можна розглядати як “чорну скриньку”. Тобто програміста може не цікавити як вона реалізована, а тільки її призначення і вхідні параметри. Тому всі необхідні параметри краще передавати через список заголовку функції. Звичайно, при необхідності можна передавати параметри і через зовнішні змінні, але це гірше, бо відноситься до побічних ефектів і програє у неточності.

Виклик функції відбувається за ім'ям **absn (d)** і на відміну від інших мов, це звертання може входити до складу виразу чи бути окремим оператором (що залежить від призначення функції).

Викликаючи функцію, зазначають фактичні параметри.

Розглянемо програму перетворення маленьких латинських літер і великі. У кодах ASCII великі літери кодуються A 65_{10} до Z 90_{10}, а маленькі від a 97_{10} до z 122_{10}. Тому перехід від маленьких до великих – відняти 32_{10}.

```
char test(char cf)
{
 if ((cf<'a') || (cf>'z')) return(cf);
 return(cf+'A'-'a');
}

main ()
{   char *st="test PrograM";
    for(a=0;a< strlen(st);a++) cout<<test(st[a]);
}
```

У цій програмі формальним параметром є символьна змінна **'cf'**, а фактичним - покажчик **'st'**. Як відбувається їхня взаємодія?

Оскільки передача параметрів відбувається за значенням, у тілі функції не можна змінити значення змінних у визваній функції, що є фактичними параметрами.

Приклад: ¶

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

Однак, якщо в якості параметра передати покажчик на деяку змінну, то використовуючи операцію переадресації можна змінити значення цієї змінної.

Приклад: ¶

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

```
        *y=k;
    }
```

При виклику такої функції як фактичних параметрів повинні бути використані не значення змінних, а їх адреси:

```
change (&a,&b);
```

Коли компілятор зустрічає виклик функції, то він робить копію значень фактичних параметрів. Ці копії передаються у функцію й привласнюються формальним параметрам. Відбуваються необхідні обчислення, по закінченню яких формальні параметри видаляються (стають невизначеними). Зрозуміло, що при цьому фактичні параметри не змінюються.

Розглянемо інший варіант цієї ж програми, коли фактичним параметром буде адреса:

```
char test2(char *cf2)
{
 if ((*cf2<'a') || (*cf2 >'z')) return(*cf2);
 return(*cf2+'A'-'a');
}

main ()
{   char *st="test PrograM";
    for(a=0;a< strlen(st);a++) cout<<test(&st[a]);
}
```

У цьому прикладі формальний параметр є покажчиком.

Отже, коли необхідно поміняти величину змінної в визваній функції або передати його із визваної функції, то необхідно передавати у функцію адресу параметра. Або передача параметрів за адресою здійснюється через покажчик. Це стосується й масивів.

Масиви як параметри функцій

Коли масиви використовуються як параметри функцій, то було б добре, щоб такі функції були придатними для масивів різної розмірності. Одним з можливих варіантів є створення динамічних масивів.

Існує наступна можливість: описати такий масив як зовнішній. Тоді у функції розміри можна не відзначати (двічі не визначати), а ще задати один або кілька параметрів - фактичні розміри масивів.

```
void arrv(int ar[], int size)
{  for(int a=0;a<size;a++) cout<<ar[a]; }

void main(void)
{ int a,arr[30];
  for(a=0;a<10;a++) arr[a]=a;
  arrv(arr,10);
   getch();
}
```

Типи функції

Правила визначення типів функції ідентичні правилам опису змінних.

Рекурсивні функції

У мові С++ передбачені посилання функції саму на себе, тобто рекурсивні функції.

Наприклад, обчислення n!

```
long factor (n)
//обчислення n- факторіалу
int n;
{ if (n==1) return (1);
  else return (n*factor (n-1));
}
```

Кожний виклик функції називається її активізацією. В даному випадку відбувається послідовність активізації, за якою слідує компілятор. Послідовність закінчується, коли **factor** поверне 1. Після цього перераховуються всі повертаємі значення до **factor (n-1)**, по якому і обчислюється n!.

Окрім рекурсивних функцій дозволяється використовувати звертання до функцій як фактичні параметри.

Наприклад

```
printf (“ %d \n”, scanf (“ %f %e %s”, &a, &b, str));
```

Тут параметром є звертання до функції **scanf**. Тому вводимо три значення – два дійсних числа та рядок. Окрім вводу, функція **scanf** повертає ціле число, яке дорівнює кількості правильно виконаних вводів. Якщо вводимо три згаданих значення через пропуск, то функція **scanf** поверне число 3, яке і буде надруковано.

Особливості побудови програм

Один з варіантів - розмістити всі функції в одному файлі. Але для більших програм краще створити кілька файлів: fil1.c, fil2.c. Тоді для їх об'єднання можна використовувати препроцесорні директиви ***#include "fil2.c".***

При відсутності формальних параметрів у дужках також потрібно визначити тип **void:** ***float fun(void)***

Особливості опису та використання фукції в ANSI-C

У початкових версіях С функція могла повертати лише одне просте значення. В ANSI-C таке обмеження зняте. Тепер функція може повертати структуру або запис. Функції НЕ можуть повертати масив або функції, але дозволяється повертати покажчик на масив або функцію.

Наприклад:

```
int matr[10][10];
char ** ptr; / / покажчик на покажчик
int * arr [10]; / / масив покажчиків
int (* vect) [10] / / покажчик на масив
```

Для визначення типу даних діють такі правила пріоритетів модифікаторів:

1. чим більш ближче модифікатор до імені, тим вищий його пріоритет;
2. для модифікаторів одного рівня [], але () мають вищий пріоритет за *;
3. круглі дужки змінюють послідовність і мають вищий пріоритет.

Тому int *st[3][5] буде означати таке:

Є модифікатор * та [3] перебувають безпосередньо біля імені, тому вони старші за [5]. Але у [3] пріоритет старший, ніж у *, тому: int *st[3][5] - масив покажчиків з трьох елементів на масив цілого типу з п'яти елементів.

Знаючи такі правила можна визначити й покажчики на функції. Вони мають слідуючий вид:

```
int (*func) (int, int);
```

Якщо круглі дужки вилучити, то int *func (int, int) означає функцію, яка передає покажчик на ціле.

Покажчики на функції реалізують процедурні змінні, масиви процедур і дозволяють передавати функції як параметри інших функцій.

Відзначимо деякі особливості роботи з покажчиками на функції:

Припустимо маємо оголошення:

```
double (*fun_ptr) (int, int);
proc (int, int);
```

Як відзначалося раніше, покажчик fun_ptr поки ще формальний параметр, тобто не посилається на конкретну функцію. Щоб зв'язати його з певною функцією, потрібно привласнити йому ім'я функції без списку параметрів:

```
fun_ptr=proc.
```

Після цього звертання до функції через покажчик буде мати вигляд:

```
(*fun_ptr) (2, 5);
```

Ясно, що конкретний покажчик можна зв'язувати й з іншими функціями відповідного характеру.

Наприклад:

```
int difference (int a, int b) { return (a-b);} // функція різниці
int sum (int a, int b) { return (a+b);} // функція суми

void main (void)
{ int (*fun_ptr) (int, int);
  int v1=15, v2=6,par;
  fun_ptr=difference;
  par=(*fun_ptr) (v1, v2);
  printf (“ par=%d \n”, par);
  fun_ptr=sum;
  par=(*fun_ptr) (v1, v2);
  printf (“ par=%d \n”, par);
}
```

Як бачимо покажчик fun_ptr тут реалізує процедурну змінну за аналогією Турбо Паскаля.

Як і звичайні змінні, покажчики на функції можуть поєднуватися в масиви.

```
int cmp_x (int, int);
int cmp_y (int, int);
int cmp_z (int, int);
```

```
int cmp_w (int, int);
int (*fcmp[4]) (int, int)={cmp_x, cmp_y, cmp_z, cmp_w};
```

Звертання до окремих функцій здійснюється як до звичайних елементів масиву:

```
int index=2;
(*fcmp[index]) (arg1, arg2);
```

Використання покажчиків на функції дає можливість задавати різні функції як фактичні параметри.

Наприклад, є стандартна бібліотечна функція:

```
void qsort (base, num, width, compare)
char * base; / / покажчик на перший елемент
int count; / / к-ть відсортованих елементів
int len; / / розмір елемента
int (* func) (); / / покажчик на функцію порівняння
```

Функція порівняння залежить від типу даних. Але вона повинна передавати ціле число:
<0 якщо перший елемент <другого;
=0 якщо елементи рівні;
>0 якщо перший елемент >другого.

У цієї функції два параметри - покажчики на 2 елемента. Коли порівнюємо рядки, то фактичний параметр:

```
extern int strcmp ();
qsort (base, count, len, strcmp);
void qsort (char *base, int num, int width, int (*compare) ());
```

Сортування дійсного масиву ***float f_mas [50];***

Потрібно зробити функцію порівняння у вигляді, необхідному для функції:

```
 qsort
int f_comp (float *x, float *y)
{ return (*x-*y);}
```

Тоді звертання до функції qsort буде мати вигляд:

```
qsort (f_mas, size, sizeof (float), f_comp);
```

Лекція 11

4.2 Структури або записи

При розв'язуванні багатьох інформаційних задач часто доводиться використовувати дані різного типу про той же самий об'єкт. Наприклад, дані про книгу: автор, назва, видавництво, рік видання, кількість сторінок. Можна було б зберігати бібліографічні дані про книги у різних масивах: автори; назви і т.п. Але це незручно. Тому виникає необхідність використання комбінованих типів, елементами яких можуть бути різні типи.
У мові С до такого типу належать структури або записи.

Опис і використання структур

Описання структур має такий вигляд:

struct < ім'я>
{ описи елементів } [ім'я змінної, …];

окремий опис елемента це фактично опис змінної – тип та ім'я.
Наприклад, для створення масиву "студенти", де про кожного треба ввести такі дані: прізвище, рік народження, стать. – можна запровадити таку структуру:

```
struct person
{ char fio[20];
  int year;
  char sex;
};
```

Ім'я структури інакше називається тегом (ярлик, етикетка). Тег дає назву структурі ті є коротким позначення тої частини структури, яка міститься в дужках.

Опис структури аналогічний опису типу у мові Паскаль, він є лише шаблоном, і не супроводжується виділенням місця у пам'яті.

У відповідності з наведеним шаблоном можна визначити змінні:

struct person stud1, stud2, stud3; або person stud1, stud2, stud3;

Зрозуміло, що область дії і шаблону, і змінних залежить від їх розташування в програмі. Якщо опис структури навести до початку функції **main**, то він діятиме на всю програму, тобто матиме зовнішній характер. Якщо ж це зробити у функції після відкриття дужки – автоматичний.

Зазначимо, що слово структура має двоїстий характер. З одного боку його відносять до опису шаблона, а з другого – до змінних-структур.

Подібно до мови Паскаль, окремий опис шаблону надає ширші можливості. На нього можна посилатись у різних функціях, тобто використовувати багаторазово.

Якщо змінні-структури визначаються лише один раз, то і опис шаблону і визначення змінних можна сумістити:

```
struct person
{ char name[20];
  int year;
  char sex;
} stud1, stud2, stud3;
```

stud1, stud2, stud3; є змінними структурного типу, тобто кожна складається із трьох елементів: прізвища, рік народження й підлога.

Оскільки структури належать до складних типів, то для них нема ні однієї операції. Тому у виразах повні імена структурних змінних використовувати НЕ можна.

В арифметичних виразах використовуються лише імена окремих елементів структури.

Ім'я елемента складається із двох частин:

< ім'я структури>.< ім'я елемента>

Окремі елементи можуть бути різного типу як простого, так і складного. Допускається використання й елементів структур.
Наприклад, якщо замість року народження впровадити дату: рік, місяць, число, то дата сама є структурою:

```
struct birth
{ int year;
  char month[10];
  int day; };
```

Тоді

```
struct person
{ char name[20];
   birth date;
  char sex; } stud1, stud2, stud3;
```

і ім’я елемента `stud1.date.day=25;`

Як і для масивів, початкові значення можна робити й для автоматичних структур

```
struct person stud1={ “Orlik”, 1972, ‘f’};
```

самі структури можуть становити інші складні типи даних, наприклад, масиви структур:

```
struct person
{ char name[20];
   int year;
  char sex;
} student[30];
```

Тоді `student[0].name=”koval”;`

На базі структур і масивів можна створювати досить складні типи даних.

Структурні змінні та покажчики

На відміну від масиву, ім'я структурної змінної НЕ є адресою. Однак в окремих випадках використання покажчиків на структуру дозволяє реалізувати ефективну програму. Наприклад, надрукувати масив даних про студентів:

```
struct person
{ char name[20];
  int year;
  char sex;
};
main ()
{ struct person stud[30], *stpr;        // stpr – покажчик на структуру
  // присвоєння значень всім елементам
```

```
  printf (“Прізвище  рік  стать\n”);
 for (stpr=stud; stpr<stud+30; ++stpr)
 printf (“ %s %d %c \n“, (*stpr).name, (*stpr).year, (*stpr).sex);
 }
```

Тут для звертання до елементів структур використані імена (*stpr).name. і насправді , при модифікації покажчика на структуру в stpr буде записана адреса наступної структури й звертання до неї можна здійснити за допомогою прямої адресації: ***stpr***. Круглі дужки тут потрібні тому, що операція **"."** має більший пріоритет, ніж ***.** І якщо дужок не буде, то вийде нісенітниця, тому що ***stpr*** - це покажчик, а не повне ім'я структури.

Бачимо, що звертання до елементів структури вийшло важким. Допускається й більш просте звертання до елементів структури через покажчик за допомогою спеціальної операції ***“->”*** *(мінус, більше)***:**

stpr->name еквівалентно ***(*stpr).name***

Тому інший варіант:

```
printf (“ %s %d %c \n“, stpr->name, stpr->year, stpr->sex);
```

У версії ANSI-C дозволено передавати структури як параметри функцій і передавати структури через ім'я функції. Крім того, для них впроваджена операція присвоєння. Інші арифметичні операції й відносини над структурами не виконуються, тому що в цьому нема сенсу.

Приклад: дії на крапками на площині. Кожна крапка задається своїми координатами (нехай буде цілими). Тоді необхідно впровадити операції над двома крапками, наприклад, додавання:

```
 struct point
 { int x;
   int y;
 };
 // додавання двох точок
 struct point addpoint (
 struct point p1, struct point p2)
 { p1.x+=p2.x
   p1.y+=p2.y
   return p1;
 }
```

Структури відіграють важливу роль у мові C++. З їхньою допомогою можна створювати динамічні структури даних, наприклад, різні списки, дерева й т.п.

Приклад:

```
// Виведення даних, використовуючи масив, покажчик і масив покажчиків на структуру
 #include <conio.h>
 #include <stdlib.h>
 #include <iostream.h>
 / / створення структури
 struct mybas
 { char fio[25];
   int year;
   char sex;
 } peole;
```

```
 mybas stud3[10]; // масив змінних типу структури
 mybas *stud5,*studuk; // покажчики на сруктуру

 void main(void)
 { int a,i;
  clrscr();
  cout<<"Масив структури"<<endl;
  for(a=0;a<10;a++) {
   for(i=0;i<10;i++)   stud3[a].fio[i]=random(26)+65;   stud3[a].fio[i]='\0';
   stud3[a].year=random(3000);
   stud3[a].sex=random(2)?'m':'f';
  }
  for(a=0;a<10;a++)                cout<<a+1<<"->"<<stud3[a].fio<<"                "<<stud3[a].year<<"
"<<stud3[a].sex<<endl;
  getch();
 // ----------------------------------------
  cout<<" Покажчик "<<endl;
  stud5=new mybas; // виділення пам'яті
  for(a=0;a<10;a++) {
   for(i=0;i<10;i++)   (*stud5).fio[i]=random(26)+65;   stud5->fio[i]='\0';
   stud5->year=random(3000);
   stud5->sex=random(2)?'m':'f';
   cout<<a+1<<" -> "<<stud5->fio<<"   "<<stud5->year<<"   "<<stud5->sex<<endl;
  }
 delete(stud5);
   getch();
 //-----------------------------------------------------
  cout<<"Масив покажчиків "<<endl;
  studuk=new mybas; // виділення пам'яті !!!!
  for(a=0;a<10;a++) {
   for(i=0;i<10;i++)   studuk[a].fio[i]=random(26)+65;
   studuk[a].fio[i]='\0';
   studuk[a].year=random(3000);
   studuk[a].sex=random(2)?'m':'f';
  }
  for(a=0;a<10;a++)              cout<<a+1<<"->"<<studuk[a].fio<<"              "<<studuk[a].year<<"
"<<studuk[a].sex<<endl;
   getch();
  delete(studuk);
 }
```

Поля

У деяких завданнях аналізу й класифікації шифр об'єкта повинен відображати наявність або відсутність певного якості. Наприклад, при синтаксичному аналізі таблиць імен змінних прийде вирішувати, до якого класу пам'яті належить змінна: автоматичного, статичного або зовнішнього. Звичайно, можна було б використовувати структуру з такими трьома елементами, а при наявності такої властивості записувати 1, при відсутності 0.

При великій кількості змінних це приведе до нераціонального використання пам'яті.

Тому краще під кожну таку ознаку можна відвести один біт і кодувати ці ознаки ступенем числа 2.

```
 #define KEYWORD 01
 #define EXTER 02
```

```
#define STATIC 04
#define AUTOM 08
```

Якщо необхідну ознаку занести до якоїсь змінної, то можна тоді використовувати порозрядні логічні операції:

```
int flag=3; // 0000 0011₂
flag=flag | STATIC;  або      flag |=STATIC
//  0000 0011.
//+
//  0000 0100
-----------------
//  0000 0111
```

Після цього в розряді №3 з'явиться одиниця. Тут незручність і, що для маніпуляції з окремими бітами потрібно підбирати спеціальну маску.
Тому для подібних завдань існує спеціальний тип даних - поля. Це тип є різновидом структури, і його опис має вигляд:

struct
{unsigned <ідентифікатор 1>: <довжина-поля 1>;
unsigned <ідентифікатор 2>: <довжина-поля 2>;
. . . . . .
} ім'я змінної;

Від шаблону звичайної структури поле відрізняється наявністю в кожному елементі :
№ - константи без знаку. Ці константи визначають, скільки біт відводиться на відповідне поле.
Після цього кожне поле має своє ім'я і з ним можна оперувати як із звичайною змінною:
flag.stat=1 або if (flag.keyword==0 && flag.auto==1) оператор;
кількість біт під те чи інше поле може бути різною

```
struct { unsigned cod1 : 2;
         unsigned cod2 : 4;
         unsigned cod3 : 8;
} pcode;
```

Між окремими полями можна робити пробіли

```
struct { unsigned cod1 : 2;
 unsigned cod2 : 4;
          : 2;
 unsigned cod3 : 8;
} pncode;
```

1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
0	0	0	0	0	0			0	0	0	0	0	0	0	0
cod1		cod2						cod3							

Тут між другим і третім полем пропущено 2 біта. Якщо всі поля не містяться в одне ціле, то переносяться в наступне. Але так, щоб поле не розривалося, а вирівнювалося по цілому.
Порожнє поле із цифрою 0 означає перехід до наступного цілого

```
struct { unsigned cod1 : 2;
```

```
        unsigned cod2 : 4;
       : 0;
        unsigned cod3 : 8;
} prcode;
```

Це означає, що третє поле буде записано з початку третього байту.

1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8
1	2	4	8	16	32	64	128		1	2	4	8	16	32	64	128
0	0	0	0	0	0				0	0	0	0	0	0	0	0
cod1		cod2							cod3							
Перший байт									Другий байт							

Приклад:

```
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
struct mybas
{ char fio[25];
  int year;
  char sex;};

struct mybas stud3[10] ;
struct mybas *stud5,*studuk;
// основна функція
void main(void)
{ int a,i;
  clrscr();
  cout<<"Масив структури"<<endl;
  for(a=0;a<10;a++) {
   for(i=0;i<10;i++)   stud3[a].fio[i]=random(26)+65;
   stud3[a].fio[i]='\0';
   stud3[a].year=random(3000);
   stud3[a].sex=random(2)?'m':'f';
  }
  for(a=0;a<10;a++) cout<<a+1<<" -> "<<stud3[a].fio<<"   "<<stud3[a].year<<"
"<<stud3[a].sex<<endl;  getch();
// ----------------------------------------
  cout<<" Покажчик "<<endl;
//  stud5=stud3;
  stud5=(struct mybas*) calloc(10,sizeof(struct mybas));
  for(a=0;a<10;a++) {
   for(i=0;i<10;i++)   (*stud5).fio[i]=random(26)+65;
   stud5->fio[i]='\0';
   stud5->year=random(3000);
   stud5->sex=random(2)?'m':'f';
   cout<<a+1<<" -> "<<stud5->fio<<"   "<<stud5->year<<"   "<<stud5->sex<<endl;
  }   getch();
// ----------------------------------------
  cout<<" Масив покажчиків "<<endl;
  studuk=(struct mybas*) calloc(10,sizeof(struct mybas));
  for(a=0;a<10;a++) {
```

```
   for(i=0;i<10;i++)   studuk[a].fio[i]=random(26)+65;
   studuk[a].fio[i]='\0';
   studuk[a].year=random(3000);
   studuk[a].sex=random(2)?'m':'f';
  }
  for(a=0;a<10;a++) cout<<a+1<<" -> "<<studuk[a].fio<<"   "<<studuk[a].year<<"
"<<studuk[a].sex<<endl;
   getch();
  free(studuk);
}
```

Об’єднання

Інколи необхідно зберігати дані різних типів в одному й тому ж місці пам'яті. Наприклад, створити таблицю, елементами якої будуть цілі, дійсні числа та символи. Значення можуть з’являтись у довільному порядку.

Для таких випадків і передбачені об'єднання, опис яких подібних до структур і має вигляд:

union имя
{ <тип> <змінна>;
…..;
} < змінна >;

Об'єднання можна розглядати як структуру, елементи якої мають 0-зміщення в пам'яті.

Звертання до елементів об'єднання таке ж, як і до структур.

Поточне значення елемента об'єднання втрачається після того, як іншому елементові буде присвоєно значення

```
union uname
{ int digit;
  double digf;
  char letter;
} fit;
fit.digit=12;
fit.digf=2.56;
fit.letter=’a’;
```

Байты							
1	2	3	4	5	6	7	8
digit							
digitf							
letter							

Зрозуміло, що всі ці значення записуються з того ж самого місця в пам'яті. Тому об'єднання має своїм значенням останнє значення.

Транслятор НЕ контролює, якого типу було останнє значення. Це повинен робити сам програміст.

Як і для структур, з об'єднаннями можна працювати за допомогою покажчиків:

```
union uname *prf;
prf -> digf=5.61;
```

Об’єднання можуть входити в структури масивів і навпаки. Наприклад:

```
struct
{   char *name;
     int flag;
     union
    { int digit;
      double digf;
      char *letter;
     } fit;
} symt[N];
```

symt[N] - це масив структур, кожна з яких складається із трьох елементів. Третій елемент є об'єднанням. Тому можна записати:

symt[i].fit.bigf=3.14;

Ініціалізувати об'єднання можна лише першим елементом, тобто в попередньому випадку - цілим значенням.

Тому, крім того, що значення елементів об'єднання записуються на те саме місце в пам'яті, усі інші властивості їх аналогічні структурам.

Слід зазначити, що реалізація об'єднань і полів залежить від особливостей архітектури ЕОМ. Тому можуть виникати питання перенесення цілих типів на різні ЕОМ.

Визначення типу

На відміну від мови Паскаль, у С++ поки ще не було можливості створювати нові типи даних, які можуть спростити програму.

Для цього можна використовувати визначення типу:

typedef <тип> <нова назва>

Наприклад, якщо прагнемо в програмі замінити тип float на ім'я REAL, то це можна зробити так:

```
typedef float REAL; // замінити тип float іменем REAL*
```

Далі в програмі можна писати:

REAL a, b, c;

Фактично нових типів тут не створюється, а надається нова назва існуючим типам.

Ясно, що те ж можна зробити за допомогою директиви препроцесора

```
#define REAL float;  /*REAL везде будет заменено на float*/
```

Однак існують і певні відмінності:

1. Визначення **typedef** виконується компілятором, а не препроцесором;
2. Функція **typedef** визначає тільки тип даних і не може визначати ім'я константі;
3. Функція **typedef** має більш широкі можливості чому препроцесор.

Наприклад:

typedef char *STRING;

Без слова **typedef STRING –** це покажчик на символ, а з ним - це ім'я типу всіх покажчиків на символ.

Тому можна далі записати:

STRING m, mp[20]; // що еквівалентно ***char *m, *mp[20];***

typedef можна вживати й для структур:

```
typedef struct
```

```
{ float re;
  float im
}; COMPLEX;
```

Тоді **COMPLEX c, d, e;** визначить структуру комплексного типу без повторень слова **struct.**

Складні імена та покажчики на функції

При відсутності параметрів у дужках потрібно визначати тип **void**

```
float fun(void)
```

Функція може повертати й структуру або запис. Функції не можуть повертати масив або функції, але дозволяється повертати покажчик на масив або функцію.

```
Наприклад:  int matr[10][10];
            char **ptr; // покажчик на покажчик
            int *arr[10]; // масив покажчиків
            int (*vect)[10] // покажчик на масив
```

Для визначення типу даних діють такі правила пріоритетів модифікаторів:
Для визначення типу даних діють такі правила пріоритетів модифікаторів:

1. чим ближчий модифікатор до імені, тим вищий його пріоритет;
2. для модифікаторів одного рівня **[]** та **()** мають вищий пріоритет за *****;
3. круглі дужки змінюють послідовність і мають вищий пріоритет.

Тому ***int *st[3][5]*** буде означати:
Модифікатори * і [3] перебувають безпосередньо біля імені, Тому вони старші за [5]. [3] старше ніж *. Тому ***st[3]*** масив, а ****st[3]*** масив покажчиків на цілий масив з п'яти елементів.
Знаючи такі правила можна визначити й покажчики на функції. Вони мають вигляд:
int (*func) (int, int);
Якщо круглі дужки вилучити, то ***int *func (int, int)*** означає функцію, яка передає покажчик на ціле.
Покажчики на функції реалізують процедурні змінні, масиви процедур і дозволяють передавати функції як параметри інших функцій.
Відзначимо деякі особливості роботи з покажчиками на функції:
Допустимо наступне оголошення:

```
double (*fun_ptr) (int, int);
proc (int, int);
```

Як відзначалося раніше, покажчик fun_ptr поки ще не посилається на ні одну функцію. Щоб зв'язати його з певною функцією, потрібно привласнити йому ім'я функції без списку параметрів fun_ptr=proc;
Після цього звертання до функції через покажчик буде мати вигляд (*fun_ptr) (2, 5);
Ясно, що конкретний покажчик можна зв'язувати й з іншими функціями відповідного характеру.

Наприклад

```
int difference (int a, int b) { return (a-b);} // функція
int sum (int a, int b) { return (a+b);} // функція
void main (void)
{ int (*fun_ptr) (int, int);
  int v1=15, v2=6,par;
```

```
  fun_ptr=difference;
  par=(*fun_ptr) (v1, v2);
  printf (“ par=%d \n”, par);
  fun_ptr=sum;
  par=(*fun_ptr) (v1, v2);
  printf (“ par=%d \n”, par);
}
```

Як бачимо покажчик fun_ptr тут реалізує процедурну змінну за аналогією мови Пасакаль.
Як і звичайні змінні, покажчики на функції можуть поєднуватися в масиви.

```
int cmp_x (int, int);
int cmp_y (int, int);
int cmp_z (int, int);
int cmp_w (int, int);
int (*fcmp[4]) (int, int)={cmp_x, cmp_y, cmp_z, cmp_w};
```

Звертання до окремих функцій здійснюється як до звичайних елементів масиву:

```
int index=2;
(*fcmp[index]) (arg1, arg2);
```

Використання покажчиків на функції дає можливість задавати різні функції як фактичні параметри.
Наприклад, є стандартна бібліотечна функція

```
void qsort (base, num, width, compare)
char *base;          // покажчик на перший елемент
int count;           // кількість впорядкованих елементів
int len;             // розмір елемента
int (*func) ();      // покажчик на функцію порівняння
```

Функція порівняння залежить від типу даних і повинна передавати ціле число:

1. <0, якщо перший елемент;
2. <, другого =0 якщо елементи рівні;
3. >0, якщо перший елемент > другого.

У цієї функції два параметри - покажчики на 2 елемента. Зрозуміло, коли порівнюємо рядки, то фактичний параметр:

```
 extern int strcmp ();
qsort (base, count, len, strcmp);
void qsort (char *base, int num, int width, int (*compare) ());
```

Сортування дійсного масиву float f_mas [50];
Потрібно зробити функцію порівняння у вигляді, необхідному для функції qsort

```
int f_comp (float *x, float *y)
{ return (*x-*y);}
```

Тоді звертання до функції **qsort** буде наступним: **qsort (f_mas, size, sizeof (float), f_comp);**

Лекція 12

4.3 Директиви препроцесора

Препроцесор С або текстовий процесор використовується для обробки тексту вихідного файлу на першій фазі компіляції. Директиви препроцесора відзначаються спеціальним знаком # (номера), який повинен бути в першій позиції відповідного рядка.
Директиви препроцесора можуть розміщатися в будь-якому місці вихідного файлу, але діють тільки для частини програми, розташованої нижче директиви.
Розглянемо деякі з них.

Директива #define

Має дві форми:

 # define ім'я текст_підстановки
 # define ім'я (список параметрів) текст_підстановки

Перша форма використовується, щоб зв'язати ім'я із часто використовуваними константами, ключовими словами, операторами й виразами:

```
#define BETA 3.56;
#define CUBE(x) ((x)*(x)*(x))
```

Імена, які заміняють константу, називаємо символічними константами, а імена, які пов'язані з операторами або виразами - макрокомандами. У тексті вихідного файлу, який іде слідом за #define всі імена заміняються на підстановку тексту. Якщо відповідне ім'я є частиною іншого, або частиною рядка, то заміна не виконується.
Якщо текст підстановки опущений, то з тексту вихідного файлу видаляється всі відповідні імена.
Корисне застосування **#define** для довгих констант, які використовуються кілька раз для звичних логічних операцій:

```
#define PI 3.141592
#define EQL ==
#define AND &&
#define OR ||
```

або для машинно-залежних констант:

```
#define INT_MAX 32767
#define INT_MIN –32768
```

Друга форма директиви зі списком параметрів передбачає, що кожне звертання до цьому імені зі списком аргументів заміняється на текст підстановки, де формальні параметри заміщаються фактичними:

```
#define ABS (x) ((x) <0 ? –(x): (x))
```

Після підстановки макрокоманди ***ABS (v)*** отримаємо підстановку ***v<0 ? – v : v*****.**
Коли текст підстановки не вміщається в одному рядку, для переносу можна використовувати зворотну дробову риску **«\»**

```
#define ZERO_ARRAY (array, size)
 { int i=0;  \
  while (i<size)  \
  array[i++]=0;  \
 }  \
```

Звертання в програмі:

```
#define MAX 50
main ()
{ int values[MAX];
  ZERO_ARRAY (values, MAX);
}
```

- це макрокоманда.

Така макропідстановка нагадує звертання до звичайної функції. Але відміна полягає в тому, що відбувається лише текстова підстановка.

Наприклад
ABS (x+z) дає (x+z) <0 ? –(x+z): (x+z).

Перевага такої підстановки полягає в тому, що вона не залежить від типу змінних. У той час, як для функції потрібно вказувати їхній тип.

Враховуючи особливості такої підстановки, для правильної реалізації потрібно використовувати круглі дужки:

Наприклад `#define sqr (x)  (x)*(x)`

Для директиви ***#define*** з параметрами можна використовувати дві препроцесорні операції:
1. Операцію ***створення*** рядка, який відбивається одним знаком #,
2. Операцію ***об'єднання*** імен, яка відображається ##.

Якщо перед формальним параметром розміщений символ #, то в результаті підстановки на цім місці буде розміщений аргумент у подвійних лапках.

Наприклад: `#define print (expr)  printf (# expr "=%f \n", expr).`
При звертанні `print (y/z);` буде текст `printf ("y/z" "=%f \n", y/z);`

Операція ## приведе до об'єднання двох аргументів в одне ім'я: якщо у визначенні використане arg1 ## arg2, то в тексті з фактичними параметрами n1, n2 буде одне ім'я **n1n2.**

Наприклад:
`#define unite (arg1, arg2)  arg1 ## arg2;` і звертання `unite (name, 2)` створить ім'я name2.

Визначення директиви **#define** може бути скасоване директивою **#undef <имя>** Це дає можливість у наступному тексті програми використовувати певне ім'я в іншому розумінні.

Наприклад
```
#define NIL (char *) 0
#undef NIL
#define NIL (float *) 0
```

Тут спочатку ім'я **NIL** заміняється нульовим покажчиком на символ, потім визначення відміняється й далі це ім'я означає нульовий покажчик на дійсне.

Директива може скасовувати обидві форми директиви ***#define***. При цьому для другої форми наводити параметри не потрібно.

Директиви умовної компіляції

Це директиви *#if, #elif, #else, #endif*, які дозволяють не компілювати окремі частини вихідного файлу після перевірки константних виражень. Програма з директивами умовної компіляції має вигляд:

#if константное выражение
[текст программы]
#elif константное выражение
[текст программы]
.....
#else
[текст программы]
#endif

умовно компілюємий текст програми починається директивою ***#if*** і закінчується обов'язково директивою ***#endif***.

Між цими директивами можуть бути розміщено кілька директив ***#elif*** (це else - if) і одна директива ***#else.***

Константні вираження НЕ можуть містити операцій sizeof, перетворення типу й констант перерахування.

```
#define SIZE 16
#include stdio.h
main()
{
char c='A';
#if SIZE==16
  int x=123;
  printf("x=%d\n",x);
#else
  static char x[SIZE]="інформатика";
  printf("x=%s\n",x);
#endif
  printf("%c\n",c);
}
```

Умовну компіляцію можна застосувати для програм, призначених для використання на різних ЕОМ та в різних ОС.

Існують і інші директиви препроцесора й деякі деталі виконання розглянутих директив, про яких можна довідатися з довідкової літератури.

Директиви компілятора або прагми

Це інструкції компіляторові, які розміщаються в певних місцях програми й використовуються для керування діями компілятора, не впливаючи на програму в цілому.

Формат директиви**:**

#pragma <последовательность символов>

Основні дерективи**:**

#pragma check_pointer ([on/off]) установлює контроль покажчиків або знімає для певної функції**;**

#pragma check_stack ([on/off]) - установлює або забороняє контроль переповнення стека;

#pragma loop_opt ([on/off]) - знімає або встановлює оптимізацію циклів окремих функцій;

#pragma message (рядок повідомлень) – посилає рядок повідомлення в stdout без припинення компіляції;

#pragma pack ([1/2/4]) – для певних структур змінює спосіб упакування елементів;

#pragma same_seg (змінна1, змінна2...) - усі наведені змінні розподіляються в тому самому сегменті даних.
Конкретний перелік наявних директив залежить від транслятора. Компілятор інтегрованої системи BC++ сприймає
check_pointer, check_stack, message, pack.

Лекція 13

Розділ 5. Файли

5.1 Особливості файлів мови С++

Як і в інших мовах, для роботи із зовнішніми обладнаннями пам'яті в мові С використовують файли.

Кількість елементів у файлі не фіксується. Кінець файлу зв'язується з іменованою константою End Of File (EOF).(-1)

Однак, на відміну від мови Паскаль внутрішня структура файлу не визначається - які саме елементи становлять файл. Уважається, що файл складається з послідовності байтів. А що саме являють собою ці байти і як їх коректно використовувати - це лежить на відповідальності програміста.

Такі поняття як "вікно" файлу, "файлова змінна" - не вживаються.

Формально файл задається покажчиком

FILE *iot; // FILE в верхньому регістрі

Однак, якщо в мові Паскаль ім'я FILE є іменем типу даних: TYPE FIL=FILE OF REAL; те в мові С++ це не так.

Точніше кажучи **FILE** - це ім'я типу структури, де описуються властивості певного файлу. Тобто **FILE** - це тип структури у файлі stdio.h з використанням типу typedef:

```
typedef struct
{ short level;  // fill/empty level of buffer - рівень буфера
  unsigned flags;       // FILE status flags - прапорці статусу файла
  char fd;       // FILE descriptor - дескриптор файла
  unsigned char hold; // ungets char if no buffer - попередній символ, якщо нема буфера
  short bsize;  //buffer size - розмір буферу
  unsigned char *buffer;       // Data transfer buffer - буфер передачі даних
  unsigned char *curp;         // Current active pointer -  поточний активний покажчик
  unsigned istemp;     // Temporary file indicator - тимчасовий індикатор файла
  short token;            // Used for validity - для перевірки коректності
} FILE
```

Тобто **iot** є покажчиком на структуру.

Звідси зрозуміло, що робота з файлами організована засобами операційної системи, а тип структури **FILE** є певною перехідною ланкою.

У мові С++ немає власних засобів роботи з файлами, а всі дії реалізуються за допомогою функцій бібліотек С++.

Ці функції поділяються на три класи:

1. верхнього рівня (з використанням терміну “потік”);

```
#include <fstream.h>
void main(void)
{
  ofstream book_file("BOOKINFO.DAT");
  book_file << " Вчимося писати програми на мові С + +, " << " друга редакція " << endl;
  book_file << "Jamsa Press" << endl;
  book_file << "22.95" << endl;
}
```

2. для консольного терміналу та порту з безпосереднім звертанням до них;

*void main(char argc, char *argv[])*

3. Нижнього рівня (з використанням терміну "дескриптор");

Поточний обмін даними

Усі дані можна розглядати як послідовність окремих байтів, або потік. Для користувача потік - це або файл на диску, або фізичне обладнання (дисплей або принтер).

Хоча потік є послідовністю окремих байтів, функції обміну для потоку дозволяють обробляти дані різного розміру й формату (від одного символу до великих структур). При цьому здійснюється форматний або безформатний обмін в буфері.

Відкриття потоків

Для виконання операцій з файлом його попередньо необхідно відкрити. Для цього використовується функція ***fopen ().*** Її параметрами є два рядки

char *name; // це ім'я файлу, наприклад «D: \ prog.c, prn»
char *mode; // режими використання файлу

fopen (name, mode)

Існує три режими доступу до файлу:

1. Читання (read):
"r"- відкрити файл для читання, файл повинен існувати;
"r+" - відкрити файл одночасно для читання й запису, файл повинен існувати;

2. Запис (write):
"w" - відкрити порожній файл для запису, якщо файл існує, він стирається;
"w+" - відкрити порожній файл для читання й запису, якщо файл існує, він стирається;

3. Доповнення (append):
"a"- відкрити файл для читання й додавання починаючи з кінця, якщо файлу немає він створюється для читання або запису (курсор наприкінці файлу).

Відзначимо, що коли файл відкривається для запису, то увесь його зміст стає недоступним, тобто в логічнім розумінні втрачається. При доповненні зміст файлу зберігається й покажчик установлюється на кінець файлу.

Оскільки параметри **name** і **mode** є рядками, то вони беруться в лапки:

```
FILE *iot;
iot=fopen (“data.txt”, “r+”)
```

Функція **fopen** повертає покажчик на структуру **FILE,** яка описує даний відкритий файл.

Однак іноді, наприклад, при спробі відкрити неіснуючий файл для читання, Функція **fopen** *цього зробити не може*. Тому, якщо файл не вдалося відкрити, функція **fopen** повертає значення ***NULL***, яке визначалося в stdio.h як 0, крім того, у глобальну змінну **errno** буде записано код помилки.

Ось чому в програмах роботи з файлами бажано перевіряти умову, чи відкрився файл, чи ні? Наприклад, так:

```
if ((iot=fopen (“data.txt”, “r+”)) ! =NULL
```

Існує два типи файлів:

t - відкрити в текстовому (перетворюючому) режимі, тобто при введенні комбінація "Повернення каретки - переклад рядка" (ПК- ПР) перетворюється до єдиного символу "перекладу рядка". При виведенні символ перекладу рядка перетворюється в комбінацію ПК-ПР;

b - відкрити у двійковому (неперетворюючому) режимі;

Якщо **t** або **b** у рядку **type** не задається, режим перетворення визначається змінної ***_fmode*** і режимом, установлюваним за замовчуванням.

Стандартні покажчики потоків

З кожною задачею автоматично пов'язується 5 потоків:

1. стандартного вводу (stdin);
2. стандартного виводу (stdout);
3. стандартного виводу повідомлень про помилки (stderr);

} потоки, зв'язані з консоллю користувача

4. додаткового потоку (stdaux);
5. стандартного потоку (stdprn).

Додатковий потік відносять до додаткового порту, до якого можна підключити допоміжну консоль, а стандартний друк – до друкуючого пристрою.

Покажчики **stdin, stdout, stderr, stdaux, stdprn** є константами, а не змінними. Тому їм не можна присвоювати інші значення покажчиків потоків.

Проте кожний з цих потоків можна переадресувати на інший пристрій або інший потік за допомогою функції freopen ("file.dat", "w", stdout)

Закривається потік для стандартного виводу і ім'я потоку **stdout** закріплюється за файлом file.dat у режимі запису. Після цього вивід у **stdout** означає виведення до файлу file.dat.

Закриття потоків

Закриття потоків здійснюється для:

- окремого потоку функцією **fclose (iot)**
- всіх потоків - функцією **fcloseall().** Остання функція НЕ закриває стандартні 5 потоків.

При закритті потоків звільнюються всі буфери, ліквідуються покажчики на файл і, якщо потрібно, підготовляється файл для зберігання.

Якщо потоки не закриваються явно, то вони закриваються автоматично при завершенні програми. Оскільки кількість відкритих потоків обмежена, то краще потоки в програмі закривати явно.

Лекція 14

Функції обміну з потоками

Робота з файлами здійснюється за допомогою різних бібліотечних функцій.

У загальному випадку у файлах може зберігатися символьна інформація (форматовані файли) і двійкова (неформатовані файли). Тому існують потоки текстові, які складаються із символів, розділених на рядки. Для розподілу на рядки використовується керуючий символ '\n'. Текстові потоки добре переносяться на інші комп'ютери, якщо в них не втримуються спеціальні символи псевдографіки фірми IBM.

Двійкові потоки - це послідовність значень типу **char**. Будь-які дані можна вважати послідовністю символів. Для визначення типу потоку при відкритті файлів додатково вказується буква **t** для текстових і **b** для двійкових потоків.

Наприклад: "r+b".

Уважається, що стандартні потоки **stdin**, **stdout**, **stderr** є текстовими, а **stdaux**, **stdprn** - ні.

Мова С++ відзначається різноманітністю функцій обміну з потоками

Об'єкт операції	Операції обміну					
	Зчитування			Запис		
	Із потоку ***stdin***	Із будь-якого потокц	Із рядку С++	В потік ***stdout***	В будь-який потік	В рядок С++
Пслідовність байт		fread			fwrite	
Окремий символ	fgetchar getchar getch	fgets getchar		fputchar putchar ungets	fputc putc	
Ціле типа int		getw			fput	
Рядок	gets	fgets		puts	fputs	
Форматовані дані	scanf	fscanf	sscanf	printf vprintf	fprintf vfprintf	sprintf vsprintf

1. Обмін символами

Для обміну символами передбачені такі функції:

```
FILE * point;
char ch;
ch = gets (point); // Зчитати символ з файлу
putc (ch, point); // Записати символ в файл
```

Коли прочитати символ не вдалося (помилка або кінець файлу), те функція повертає -1. Функція **fgets** і **fputc** діють аналогічно функціям **gets**, **putc**, але реалізовані інакше.

2. Обмін рядками

fgets (string, MAXLIN, point) - читання рядка з файлу з покажчиком **point string** - ім'я рядка, куди зчитуються символи рядка; ***MAXLIN*** - максимальна кількість символів, яку потрібно прочитати. ***третій параметр*** - це покажчик point на файл, звідки здійснюється зчитування, Функція ***fgets*** припиняє роботу, якщо прочитане ***MAXLIN-1*** символів, або до зчитування символу закінчення рядка **('\n')**. Повертає ***NULL***, якщо натрапили на кінець файлу. ***Status=fputs(string, point)*** - запис рядка у ***файл.*** Якщо відбулася помилка або кінець файлу, то вертається ***EOF.***

3. Форматний обмін

fscanf (point, "%d", &psi) – зчитування з файлу. Функція передає кількість правильно прочитаних значень, а також EOF, коли файл вичерпано

fprint (point, "psi=%d \n", psi***)*** - запис в файл

У файл **point** записується ціле значення **psi**. Функція передає кількість правильно записаних значень.

Для обміну цілими передбачено дві функції **getw(point)**, **putw(int, point)**;

```
int c;
FILE *point;
c=getw (point);
putw (c, point);
```

Функція **getw** зчитує два байти з потоку **point** і передає змінний c. Функція **putw** записує ціле значення змінної з у потік **point**. Але для різних систем може виникнути питання перенесення. Передбачено дві функції для обміну блоками байтів:

1. **fread (buffer, size, count, stream);**

```
struct mybas
{ char fio[25];
  int year;
  char sex;};
mybas stud3[NUM], stud5 ;
FILE *filw,*filr;

void main(void)
{ int i;

   fwrite(&stud3[i],sizeof(stud3),1,filw);
   fread(&stud5,sizeof(stud5),1,filw);// !!!!!
```

З потоку ***stream*** зчитується count блоків, кожний з яких має розмір ***size***, у буфер ***buffer***. Функція передає дійсну кількість лічених блоків і ніяких форматних перетворень НЕ відбувається.

2. ***fwrite(buffer, size, count, stream);***

З буфера ***buffer*** записується count блоків, кожний з яких розміром ***size*** у потік stream. Функція передає кількість у дійсності записаних блоків. Відзначимо ще дві функції форматного обміну. Функція ***sscanf*** відрізняється від функції ***fscanf*** тим, що зчитування здійснюється з рядка string:

sscanf(string, format_string, list);

Функція sprintf відрізняється від ***fprintf*** тем, яке записує значення змінних не в потік, а в рядок:

sprintf(string, format_string, list).

Отже першу функцію можна використовувати для перетворення символів у внутрішню виставу чисел, а другу навпаки - із внутрішньої вистави до символьного.

```
char string[10];
float value=3.12;
sprintf (string, “%f”, value);
```

У рядку string одержимо символьну виставу числа 3.12 у формі з фіксованою крапкою.

```
char string[10]=”3.12”;
float value;
sscanf (string, “%f” &value);
```

Рядок "3.12" перетвориться у внутрішню виставу в змінній value.

Приклад: читати інформацію з файлу цілих, записувати в змінну t і виводити на екран. Алгоритм: якщо файл відкрився, то в циклі зчитувати значення в змінну ***t***, поки функція ***fscanf*** не дорівнює **EOF**, і виводить значення змінної ***t***. Після цього файл закрити. Якщо файл не відкрився, то вивести повідомлення: “ **файл не відкрито** ”:

```
main ()
{ int t;
  FILE  *fp;
   if (( fp=fopen (“DATA.TXT”, “r” )) ! =NULL)
          { while (fscanf (fp, “%d” & t) ! =EOF)
             printf (“число t =%d \n”, t);
             fclose (fp);
          }
  else printf (“файл не відкрито”);
  getch ();}
```

Довільний доступ до потоку

У наведені вище прикладах потоки використовувалися як файли послідовного доступу. Операції читання або записи в потік починалися з поточної позиції потоку, яка визначається внутрішнім покажчиком потоку. При відкритті покажчик показує на початок потоку в режимі **"r"** і **"w"**, і на кінець потоку в режимі **"a"**.

Після виконання певної операції обміну покажчик змінюється й рівняється нової поточної позиції потоку. Наприклад, якщо з потоку прочитано 1 символ, то покажчик збільшується на 1.

Але існує можливість позиціонувати покажчик на будь-яку позицію в потоці, тобто реалізувати прямий доступ до файлу. Існує п'ять функцій для позиціонування покажчика потоку:

1. **ftell(point)** - передає поточну позицію покажчика потоку:

```
long ipos;
ipos=ftell(fp);
```

2. fgetpos(point, long) – передає поточну позицію покажчика потоку, передає 0, якщо визначення відбулося правильно й -1 інакше:

```
long ipos;
fgetpos(fp &ipos);
```

3. fsetpos(point, long) – установлює відповідну позицію покажчика потоку;

```
long ipos;
      fsetpos (fp, &ipos);
```

4. fseek(fp, count, start);, установлення покажчика на будь-яку позицію покажчика потоку, де **fp** - покажчик на файл; **count** - кількість байт типу **long**, визначає абсолютну або відносну позицію у файлі; **start** - типу **int**, з якої позиції потрібно відраховувати кількість байт **count**:

start:
- 0 – від початку файла — SEEK_SET
- 1 – від поточної позиції — SEEK_CUR
- 2 – від кінця файла — SEEK_END

```
Наприклад:
fseek (fp, 0L, SEEK_SET) // на початок потоку
fseek (fp, n, SEEK_CUR) // на n байт від поточної позиції
fseek (fp,-n, SEEK_CUR) // на n байт до поточної позиції
fseek (fp, 0L, SEEK_END) // на кінець потоку
```

5. ***rewind (fp)*** перейти на початок файлу, аналогічна ***fseek (fp, 0L, SEEK_SET).***

Якщо потік відкрито для додавання (режим **"a"**, **"a+"**), тоді запис завжди здійснюється в кінець файлу. Тобто хоч відповідними функціями можна знайти потрібну позицію, але при спробі запису покажчик попередньо буде встановлений на кінець файлу. Відзначені функції організації прямого доступу не можна застосовувати до стандартних файлів в текстовому форматі.

Лекція 15

Керування буферизацією

Потоки по умовчуванню є буферизуємі, тобто при відкритті потоку з ним автоматично зв'язується область пам'яті, яку називають буфером. При читанні даних з потоку інформація розміщається у вхідному буфері й дані зчитуються з буфера. Коли весь буфер оброблений, у буфер зчитується наступний блок даних.

При записі зміст вихідного буфера записується у відповідний потік (буфер звільняється) коли буфер заповнений, або закривається відповідний потік, або коли програма успішно завершена.

Це підвищує ефективність програми, тому що обмін реально здійснюється не одним елементом даних, а цілими блоками. Буфери, створювані операційною системою, недоступні для користувача.

Але існує можливість створювати не системні буфери для обміну, або робити обмін небуферизуємі. Це можна здійснити за допомогою функцій:

1. ***void setbuf(stream, buffer);***
2. ***int setvbuf(stream, buf, type, size)***.

0 – виділено;

<>0 – Полилка при виділенні

З цими буферами можна поводитися як зі звичайними змінними.

Функція **setbuf** зв'язує потік з буфером, який повинен бути масивом символів довжиною **BUFSIZE** (ця константа у файлі **stdio.h** і дорівнює 512).

Наприклад**:**

```
#include <stdio.h>
FILE *fp;
char my_buf [BUFSIZE];
fp=fopen (“d:\data.txt”, “r+”);
setbuf (fp, my_buf);
```

Далі потік використовує буфер **my_buf**. Якщо потрібно зробити обмін без буферизаціїі, то замість імені буфера потрібно поставити ненульовий покажчик **NULL**.

```
setbuf (fp, NULL);
```

Функція **setvbuf** дозволяє встановити довільний розмір буфера, але НЕ більше 65535 байт ***(64К).***

```
Наприклад
FILE *fp;
char *buf;
unsigned n;
setvbuf (fp, buf, _IOFBF, n);
```

Функція **setvbuf** дозволяє користувачеві керувати буферизацією і розміром буфера для потоку **stream**. **Stream** може посилатися на відкритий файл. Масив, на який указує **buf**, використовується як буфер, якщо він не є **NULL**, тобто потік не є буферизованим. Якщо потік буферизований, використовується тип, вказаний по type; цей тип може бути або **_IONBF**, або **_IOFBF**, або **_IOLBF**. Якщо використовується тип **_IOFBF**, розмір буфера визначається по **size**; якщо використовується тип **_IOLBF** або **_IONBF**, потік є НЕбуферизованим, а **size** і **buf** ігноруються.

_IONBF Буфер НЕ використовується, незважаючи на присутність **size** і **buf**;
_IOFBF ПОВНА буферизація, якщо **buf** не є **NULL**; тому **buf** використовується в якості буфера, а **size** - його розміру;

_IOLBF Аналогічно _IOFBF (строкова буферизація).

setvbuf (fp, buf, _IONBF, n); // не буферизуется

За допомогою цієї ж функції можна збільшувати розмір системного буфера:
setvbuf (fp, NULL, _IOFBF, n);– размер увеличивается к n байтам.

Коли програма завершується аварійно, те вихідний буфер може бути не вивантаженим, що може привести до втрати інформації.
Для вивантаження буфера можна використовувати функцію:
int fflush (FILE *stream)

Якщо заданий потік **stream** відкритий для виводу, то вміст буфера, пов'язаного з потоком **stream** функції **fflush**, записується у відповідний файл. Якщо потік відкритий для введення, то функція **fflush** очищає вміст буфера. Після виклику функції потік залишається відкритим. Для небуферизованого потоку застосування цієї функції не ефективно.

int flushall() **-** функція записує вміст усіх буферів, пов'язаних з відкритими **input** потоками, у відповідні файли. Усі буфери, пов'язані з відкритими потоками, очищаються; наступна операція читання (якщо вона є) зчитує нові дані із вхідних файлів у буфер. Після виклику функції **flushall** усі потоки залишаються відкритими.

Дана функція повертає кількість відкритих потоків (вхідних і вихідних). У випадку помилки значення, що вертається, НЕМАЄ.

Слід зазначити, що потрібно обов'язково закривати файли, робота з якими проводиться через буфери користувача.

Текстовий и двійковий режим обміну

Як відзначалося, можна відкривати потік у текстовому або двійковому режимі. Але ці режими мають певні особливості.

У текстовому режимі при введенні комбінація символів "повернути каретку - перевести строку" (\015 \012) перетвориться в один символ - перевести рядок. При висновку символ "перевести строку" перетвориться у два: \015, \012. Крім того, комбінація клавіш Ctrl+z (0x1a) сприймається як кінець файлу при висновку.

У двійковому режимі перетворення символів ***"ПК - ПС"*** не здійснюється.

Режим обміну з потоком можна змінювати після його відкриття. Для цього призначена функція нижнього рівня ***setmode***. Як відзначалося на рівні потоків функції використовують покажчик потоку.

На нижньому рівні використовується інше поняття - дескриптор файлу ***(handle)***. Це є ціле число, яке зв'язується з файлом і використовується для посилань на даний файл. Так стандартні потоки ***stdin, stdout, stderr, stdaux, stdprn*** мають відповідні дескриптори ***0,1,2,3,4.***

int setmode (int handle, unsigned mode);(якщо функція повертає 0 - нормально, інакше –1)

handle – дескриптор файла, ***mode*** – режим.
Режим задається двома константами в файле <**fcntl.h**>
O_BINARY двійковий
O_TEXT текстовий.

Отже, перед використанням функції setmode потрібно одержати дескриптор певного файлу, що й виконує функція

fileno ***(FILE *ptr)***, параметром якої є покажчик файлу.

Отже перехід від текстового до двійкового режиму можна здійснити так**:**

```
# include <stdio.h>
# include <io.h> //  для функції
```

```
# include <fcntl.h> // setmode
FILE * ptr;
if (setmode (fileno (ptr), O_BINERY) == 0)
printf ("Двійковий режим встановлено \ n");
else prinrf ("Двійковий режим встановити не вдалося \ n");
```

Інші функції обробки потоків

Для роботи з потоками можна використовувати й деякі інші функції.
Для визначення кінця файлу призначена функція:

int feof (FILE *stream), яка передає нуль, якщо досягнуто кінець файлу при спробі прочитати символ, що випливає за останнім. Інакше її значення не нуль.

int ferror (FILE *stream) дозволяє встановити причину помилки читання або запису. Якщо 0 - помилки не було. Якщо трапилася помилка, то внутрішній індикатор помилки файлу буде встановлено доти, поки потік не буде закритим або **rewind**, або **clearrer**.

void rewind (stream), або **void cleareгr(stream**) - яка записує в внутрішній індикатор помилки файлу нуль. Це ж здійснюється й при закритті потоку.

Змінити назву файлу можна за допомогою функції:
int rename (char *oldname, char *newname) і поміняти **oldname** на нову назву **newname**. Якщо така процедура здійснилася успішно, то функція передає нуль. Цю функцію можна використовувати для запису файлу до іншого каталогу, але в межах одного обладнання.

Знищити файл можна за допомогою функції:
int remove (const char *filename). Наступна спроба відкрити файл із таким іменем буде помилкою.
int fstat(handle,buffer) - одержує інформацію про відкритий файл, пов'язаний з даним ***handle***, і запам'ятовує її в структурі, на яку вказує **buffer**.
buffer - структура, тип **stat** якої оголошений в <sys.h \ stat.h>, містить наступні поля:

Поле	Значення
st_mode	Бітова маска для інформації про режим файлу. Біт **S_IFCHR** установлюється, якщо handle посилається на обладнання. Біт **S_IFREG** установлюється, якщо handle посилається на звичайний файл.
st_dev	Номер обладнання диска, щомістить файл, або **handle** - у випадку іншого обладнання.
st_rdev	Номер обладнання диска, щомістить файл, або **handle** - у випадку іншого обладнання (аналогічно **st_dev**).
st_nlink	Завжди 1.
st_size	Розмір файла в байтах.
st_atime	Час останньої модифікації файла.
st_mtime	Час останньої модифікації файла. (аналогічно **st_atime**).
st_ctime	Час останньої модифікації файла. (аналогічно **st_atime** та **st_mtime**).

Приклад.

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>
 stat buf;
int fh, result;
fh=open("tmp/data", O_RDONLY); // дескриптор файлу
result=fstat(fh,&buf);
if (result==0)
   printf("file size is %ld\n",buf.st_size);
```

Лекція 16

5.2 Обмін нижнього рівня

На відміну від обміну з потоками, функції нижнього рівня здійснюють небуферизуємий і неформатовиний обмін і безпосередньо викликають засоби обміну операційної системи. Як відзначалося, на нижньому рівні з файлом зв'язується його дескриптор. Інтерфейсним для функції нижнього рівня є файл, який потрібно підключити до відповідної до програми.

Відкриття та закриття потоків

Перед обміном з файлом його треба відкрити за допомогою одної з таких функцій:

fd=creat(name, pmode) – створити новий файл

fd **-** дескриптор файлу, через який можна буде звертатися до файлу;
pmode:
S_IREAD – дозволено зчитування (0)
S_IWRITE – дозволено запис (1)
S_IREAD | S_IWRITE – дозволені запис та зчитування (2)

int fd=open(pathname, oflag [[,pmode]] – відкрити файл***)***
char *pathname – ім'я файла
int oflag – дозволено тип операції
int *rwmode* – дозволено тип доступу
oflag:
O_BINARY – в двійковому режимі;
O_TEXT – в текстовомц режимі;
O_RDONLY – тільки для зчитування;
O_WRONLY – для запису.
O_RDWR – для зчитування і запису;
O_TRUNC – відкривається існуючий файл і весь зміст знищується;
O_APPEND – для запису в кінець файлу;
O_CREAT - створюється новий файл для запису; це не ефективно, якщо існує файл, обумовлений по **pathname** –імені.
O_EXCL- вертається значення помилки, якщо існує файл, обумовлений по **path**- імені. Застосовується тільки разом з O_CREATE.
Якщо потрібно використовувати кілька констант, то вони записуються з використанням операції побітового "АБО":

```
handle =open ("data", O_CREAT | O_WRONLY);
```

```
Наприклад:
# include <stdio.h>
# include <io.h>
# include <fcntl.h> // для визначення констант
int handle;
if ((handle = open ("data", O_RDONLY)) == 1)
  printf (stderr, "Помилка відкриття файлу \n");
```

int sopen (pathname, oflag, shflag [,pmode]) – відкриває файл для загального використання й підготовляє його до наступного розділеного читання або запису, що визначається значенням ***oflag*** або ***shflag***.

***char *pathname* –** ім'я файлу
int oflag - тип дозволених операцій
int shflag - дозволений тип розділення
int pmode дозволений тип доступу

shflag:
SH_COMPAT - встановлюється режим сумістності
SH_DENYRW - доступ по читанню й запису у файлі НЕ дозволено
SH_DENYWR - доступ по запису у файлі НЕ дозволено
SH_DENYRD - доступ по читанню у файлі НЕ дозволено
SH_DENYNO - доступ по читанню й запису дозволено

int chmod(pathname,pmode) - змінює дозволений доступ для файлу, заданого **path**-іменем. Якщо дозвіл на запис не задано, файл доступний тільки для читання. В MS DOS усі файли доступні для читання, тому не можливо задати дозвіл тільки на запис. Тому режими **S_IWRITE** і **S_IREAD|S_IWRITE** є еквівалентними.

Зчитування та запис даних

Для читання й запису інформації призначено дві функції **read** і **write**. Їх формат:

read (fd, pointer, size);

write (fd, pointer, size);

fd – дескриптор файлу, який був визначений функціями **create** абоі **open;**
pointer – покажчик на буфер даних, до якого записується (зчитується) інформація**;**
size – довжина цього буфера.

Якщо обмін успішний, то функції передають кількість фактично переданих байт. Коли помилка або кінець файлу, то функції одержують значення EOF (-1).

Приклад: до одного файлу дописати зміст іншого**.**

Приклад:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <conio.h>
void main (void)
{ int hand1, hand2, size;
  char block[512] *file1, *file2;
  puts (“Ввести имена \n”);
  gets (file1);  gets(file2);
  printf (“ %s %s  \n”, file1, file2);
  if ((hand1 = open (file1, O_RDONLY)) == -1)
   {printf ("Помилка файлу % s \ n", file1); return 1;}
 if ((hand2 = open (file2, O_APPEND | O_CREATE)) == -1)
    {printf ("Помилка файлу % s \ n", file2); return 1;}
printf ("hand1 = % d hand2 = % d \ n", hand1, hand2);
while ( (size = read (hand1, block, sizeof (block) ) )! = EOF )
{size = write (hand2, block, size);}
close (hand1);
close(hand2);
getch();
}
```

Прямий доступ до файлів

Функції **read** і **write** здійснюють обмін у режимі послідовного доступу. При необхідності, можна реалізувати й прямий доступ до файлу. Для позиціонування файлу використовується функція:

lseek (file_desc, no_bytes, start)

file_desc – дескриптор файлу;
long no_bytes – задає абсолютну або відносну позицію байта у файлі;
start – ціле значення:
0 – рахувати від початку файла;
1 – рахувати від поточної позиції;
2 – рахувати від кінця файлу.
Як бачимо, формально функція **lseek** відрізняється від **fseek** - тільки першим аргументом - дескриптором файлу.

long tell (int handle) - повертає поточну позицію, на яку встановлено покажчик файлу.

filelenght (handle) - повертає кількість байтів у файлі.

unlink (file_specification) - видалення файлу

Наприклад:
```
unlink ("C:\TEST.TXT");
```

Функція ***unlink*** повертає:
0 - файл успішно видалено;
-1 свідчить про помилку.
Відзначену функцію можна використовувати тільки тоді, коли файл НЕ відкрито.

Функції обміну з консоллю

Для небуферизованого обміну з консоллю призначено такі функції:

cgets - введення рядка з консолі;

cputs - вивести рядок на консоль;

getch - введення символу з консолі без відображення;

getche - введення символу з консолі із зображенням;

putch - вивести символ на консоль;

cscanf (format-string[,argument...]) - форматоване введення з консолі;

cprintf (format-string[,argument...]) - форматоване виведення на консоль;

kbhit - перевіряє, чи була натиснута клавіша й повертає ненульове значення, якщо клавіша була натиснута. А якщо ні, то вертається 0.
Прототипи відзначених функцій наведені в інтерфейсному файлі **<conio.h>**.

Розглянемо програму визначення коду символу, прочитаного із клавіш. Справа в тому, що клавіатура посилає до процесора так званий "скан- код" клавіші. Кожна клавіша має свій скан- код. Наприклад 1 - має код 1, 2 - код 2 і т.д. до 83. До процесора надходить 2 сигнали: при натисканні клавіші - скан- код, а при відпусканні - код відпустки, яка рівняється 128+ скан-код. Уважається, що клавіша натиснута, поки не отримано скан-код відпускання.

```
# include <stdio.h>
# include <conio.h>
main ()
{char c;
// цикл, поки не прочитана комбінація клавіш CTRL + Z /
  while ( (c = getch ()) != 'x1a')
  if (c == 0 || c == 'xe0 ") // ' xe0 '- 224
   printf ("код спец клавіші або комбінації клавіш -% lx \n", getch ());
else
printf ("код прочитаного символу -% lx \n", c);
}
```

Функція **kbhit** перевіряє, чи натиснута клавіша, передає:
0 - коли клавішу не нажали
ціле значення нажатої клавіша.
Її можна використовувати для організації циклів, вихід з яких відбувається при натисканні клавіші:

```
while ( ! kbhit) оператор;
```

Якщо треба просто затримку: `while ( ! kbhit);`

Інші функції обміну

Крім відзначених функцій обміну існують функції, які безпосередньо використовують системні виклики MS DOS:

_dos_open - відкрити файл;
_dos_create - створення нового файлу;
_dos_close - закрити файл;
_dos_read - читати файл;
_dos_write - запис до файлу.
Прототипи цих функцій наведені в в інтерфейсному файлі **dos.h**. У цьому ж файлі є прототипи функцій для обміну з портами:

int inport (int port), int inportb (int port) - для введення слова (або байта) з порту номер **port**.
void uotport (int port, int word), void uotportb (int port, char byte) - для виведення слова (або байта) у порт із номером **port**.
Дві перші функції повертають уведене слово або байт. Отже, розглянули функції обміну 4 рівнів:
1) буферизуємого на рівні потоків;
2) нижнього на рівні дескрипторів;
3) консольного небуферизуємого;
4) обміну за допомогою функцій переривань MS DOS.

Доцільність використання певних функцій залежить від кінцевої мети призначення програми. Буферизуємий обмін забезпечує найбільшу вірогідність перенесення програм.

Буферизуємий обмін забезпечує високу надійність обміну: після того, як увели дані, їх можна прочитати на екрані й при необхідності виправити. Тільки при натисканні клавіши Enter інформація буде видалена з буфера. Ясно, що швидкодія буде нижче й обсяг пам'яті більш високий. Функції нижнього рівня мають гірший покажчик мобільності.

Обмін з консоллю характеризується швидкістю, що може бути важливим в ігрових програмах. Ясно, що функції нижнього рівня й консольні займають менше місця в пам'яті.

Але доступ до файлу за допомогою функцій різного призначення може привести до втрати інформації.

Тому, наприклад, функції обміну за допомогою засобів MS DOS не потрібно використовувати разом з функціями обміну потоками, нижнього рівня або з консоллю.