Київський національний університет імені Тараса Шевченка

Факультет радіофізики, електроніки і комп'ютерних систем

МЕТОДИЧНИЙ ПОСІБНИК З КУРСУ «ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ. Мови C/C++»

Київ – 2021

Рецензенти:
канд. фіз.-мат. наук, доц.Іваненко Д.О.,
канд. фіз.-мат. наук, доц. Кононов М.В.

Єфіменко С.В.

Методичний посібник з курсу «Об'єктно-орієнтоване програмування. Мови С/С++»

Методичний посібник призначений для студентів факультету радіофізики, електроніки і комп'ютерних систем для вивчення курсу «Об'єктно-орієнтоване програмування». Посібник повністю відповідає програмі курсу та містить повне викладення курсу.

Затверджено вченою радою факультету радіофізики,
електроніки і комп'ютерних систем
(протокол № 8 від 14 грудня 2021 року)

Вступ

Курс «Об'єктно-орієнтоване програмування» наразі є другим курсом програмування, який вивчається студентами першого курсу факультету радіофізики, електроніки та комп'ютерних систем, після знайомства з мовою C# в курсі «Основи програмування". Початкові поняття, які вживаються в програмуванні, а також принципи двійкової арифметики, необхідні для розуміння внутрішнього представлення інформації, викладені в посібнику [1]. Вони є необхідними для якісного вивчення питань роботи з динамічною пам'яттю та адресацією, які в свою чергу є наріжним каменем як в мові С, так і в С++. Хочеться сподіватись також, що уявлення про класи, одержане при вивченні мови C#, зараз буде розвинуте та поглиблене.

Слід зазначити, що в даному курсі спочатку викладені основи класичної мови С з поступовим використанням можливостей мови С++. І оскільки мова С вкладається в С++, як частина пазлу, такий підхід виглядає цілком природним. На завершення курсу ми познайомимось із деякими принципами метапрограмування, реалізованими шаблонами класів мови С++.

Автор буде вдячним всім, хто запропонує свої пропозиції задля покращення даного посібника.

Декілька фактів з історії мов С та С++

Творцем мови С є Деніс Рітчі, який, працюючи в фірмі Bell Labs, в 1972 році розробив нову мову програмування та перший компілятор з неї. До речі, він же був одним із ключових розробників операційної системи Unix. Ця мова стала результатом еволюційного розвитку мов B та BCPL, які використовувались для реалізацій операційних систем та мовних процесорів, завдяки своїм можливостям забезпечення ефективного інтерфейсу з апаратними засобами. З назвою мови автор не переймався – просто «С» за правилом абетки. Основною відміною та безперечною перевагою нової мови перед своїми безпосередніми попередницями стало введення типів даних. В 1978 році була видана книжка Б. Кернігана та Д. Рітчі з повним описом мови, яка стала бестселлером та не складає своїх позицій і дотепер. В 1983 році був затверджений ANSI-стандарт мови С, що сприяло ще більшій її популярності. Відтоді пройшло декілька десятків років, а мова С як і раніше використовується багатьма програмістами.

У зв'язку із виникненням нової парадигми програмування – об'єктно-орієнтованого програмування, неодноразово мали місце спроби створити об'єктно-орієнтовану мову, яка була б спадкоємицею мови С. Серед декількох варіантів найбільшу ринкову спроможність одержала мова С++ (дотепна назва, адже, як ми знаємо, два плюси – позначення операції інкременту, тобто просування вперед на одиницю), Цю мову Б'ярн Страуструп, співробітник тієї ж фірми Bell Labs, почав розробляти з 1979 року. Причиною популярності мови С++ стало саме те, що мова С++ сумісна з мовою С знизу. Тобто всі програми, написані мовою С, можуть виконуватись під управлінням компілятора С++. Прийняття в 1998 році ANSI-стандарта мови С++ зробило її позиції досить вагомим серед інших сучасних об'єктно-орієнтованих мов програмування. Відтоді було прийнято багато поправок до стандарту, зрозуміло, що не всі вони будуть враховані у даному тексті, тим більше що розвиток програмування триває, з'являються нові задачі, нові підходи і нові мови програмування.

Давайте пригадаємо, який зміст вкладається у парадигму об'єктно-орієнтованого програмування.

Якщо в структурному (модульному) програмуванні визначальним фактором були алгоритми обробки даних, саме вони диктували зв'язки між модулями, то в об'єктно-

орієнтованому програмуванні на перший план виходять класи, вони визначають структуру програми, вони забезпечують «спілкування» окремих частин програми. Класи є уособленням об'єктів, що існують в оточуючому світі, абстрактною копією їх. Класи можуть приховувати нюанси свого функціонування, інкапсулюючи в собі інформацію і залишаючи користувачеві лише можливості використання, закладені у відкритому інтерфейсі. Класи дозволяють спадкування – створення нових класів на основі вже створених з використанням частини їх функціоналу і з доповненням його новими можливостями. І нарешті – поліморфізм, здатність об'єктів похідного класу демонструвати реакцію, відмінну від закладеної у базовий клас, на однакові запити. Ось ці «три кити»: ***інкапсуляція, спадкування та поліморфізм*** складають основу ***концепції об'єктно-орієнтованого програмування***.

Таким чином, мова С – досить революційна для свого часу, слугувала ідеї структурного програмування, а мова С++ виникла на підтримку об'єктно-орієнтованого програмування.

Лекція 1. Знайомство з мовами С/С++. Основні стандартні типи. Операції.

1. Перша програма

Для початку знайомства виконаємо першу програму мовою С. Чого можна вимагати від першої програми? Традиційно ми хотіли б, щоб вона після свого запуску вивела деякий текст на екран. Зауважимо, що вміст частин рядків, які починаються двома знаками `//`, являють собою коментарі, тобто пояснення до коду, які ігноруються компілятором. Так само, як і вміст одного або більше рядків обмежених символами `/*` та `*/`. Рядки, що починаються символами **#**, є так званими директивами препроцесору – програми, яка обробляє код до початку роботи компілятора. Конкретна директива **`#include`** означає команду підключити так звані заголовочні файли (header-files), які містять потрібні для роботи програми функції або константи. Назва **`main`** – ідентифікатор функції, яка запуститься першою при виконанні будь-якого програмного коду. Ось код цієї першої програми:

```
/**********     Перша програма мовою С     **********/
#include <stdlib.h> // Файл деяких службових функцій
#include <stdio.h>  // Підключення файлу підтримки вводу-виводу
int main ()              // Головна функція
{
// printf - функція для виводу на екран
printf ("Our first program");
system ("PAUSE");   // Затримка – функція із stdlib.h
return 0;                // Завершення роботи функції main ()
}
```

Результатом запуску на виконання цієї програми стане виведення на екран повідомлення **«Наша перша програма»**, після чого програма перейде у стан очікування введення будь-якого символу.

Цей самий код успішно буде виконаний і під керівництвом будь-якого компілятора мови С++. Проте давайте оцінимо, як можна було досягти аналогічного результату засобами саме С++.

```
/**********     Перша програма мовою С++      **********/
#include <cstdlib>  // Файл деяких службових функцій
```

```
//Підключення файлу підтримки потоків вводу-виводу
#include <iostream>
using namespace std; // Інструкція для закріплення стандартного
// простору імен – забезпечує коротке звертання до стандартних
// потоків вводу-виводу
int main ()                 // Головна функція кожної програми
{
cout << "Our first program" << endl;//
     system ("PAUSE");      // Затримка – функція із stdlib.h
return 0;                   // Завершення роботи функції main ()
}
```

Що змінилось в цьому фрагменті порівняно з попереднім? По-перше, змінились назви заголовочних файлів, вони тепер не містять суфікс «**.h**» (можна використати , як раніше ідентифікатор **stdlib.h**, а не перейменований файл **cstdlib** – компілятор просто зробить зауваження щодо використання архаїчних імен). По-друге, змінились засоби принаймні виведення інформації – в C++ для виведення інформації використовують стандартний потік **cout** (для введення – **cin**), в який операцією << просто відправляємо елементи інформації або керуючі символи чи маніпулятори. Втім, і функція **printf**, як і раніше успішно виконала би виведення – ми ж пам'ятаємо, що C сумісна знизу з C++.

Будемо вважати тепер себе досить вмотивованим, мобілізованим та сміливими і перейдемо безпосередньо до знайомства з мовами C та C++.

2. Алфавіт та лексеми

Алфавіт мов C та C++ включає латинські літери (великі та малі), цифри, певні символи (наприклад, для позначення операцій або пунктуації). Програма мовою C/C++ являє собою, як можна побачити вище, послідовність рядків символів, які об'єднані в так звані ***лексеми*** (tokens), тобто основні одиниці мови програмування, які мають самостійний смисл (простіше кажучи, те, що в звичайній людській мові зветься словами).

До лексем мови входять:

- ключові (зарезервовані) слова (key words);
- операції (operators);
- ідентифікатори (identifiers);
- обмежувачі (тобто розділові знаки, такі як: **( ) { } [ ] : ;**);
- константи, або ж літерали (literals).

Ці елементи мови відокремлюються один від одного коментарями, пробілами або деякими спеціальними символами, які відносяться до групи так званих «пробільних літер», наприклад знаки табуляції або початку нового рядка.

Коментарі, як вже зазначалось вище, обмежуються символами **/*Це коментар*/**, не аналізуються компілятором і використовуються як пояснення до тексту програми. Рядкові коментарі починаються двома знаками **//** (слеш) і продовжуються до кінця поточного рядка, дозволяючи пояснювати текст безпосередньо в лядку.

Ключові (зарезервовані слова) – слова, які сприймаються компілятором тільки в певному контексті і не можуть бути використані з іншою метою.

Операції бувають односимвольні, як, наприклад, звичні нам знаки арифметичних операцій + - * / , або двох- та навіть трьохсимвольні. Повний перелік операцій буде розглянуто нижче.

Ідентифікатори (інколи кажуть просто імена) використовуються для позначення констант, змінних, функцій, файлів, тощо, створених програмістом. Складаються з літер латинського алфавіту, цифр та знаку підкреслення, який вважається літерою і дозволяє зробити ідентифікатор більш наочним, наприклад, **`Max_Len_of_Vector`**. Але починається ідентифікатор завжди з літери (в тому числі і із знаку підкреслення _), при цьому маленькі і великі літери розрізняються (тобто n і N – різні ідентифікатори). Слід зауважити, що різні компілятори сприймають різну довжину ідентифікаторів, але ANSI-стандарт передбачає тільки 31 значущий символ.

Константи в мовах С та С++ бувають чотирьох типів.

1. Цілі константи можуть бути задані в десятковій, вісімковій або шістнадцятковій системах числення:

А) десяткові константи складаються із десяткових цифр від 0 до 9, причому перша не може бути нулем, наприклад: 100, 1024, -1, 0;

Б) вісімкові константи завжди починаються з 0 і включають цифри від 0 до 7, наприклад: 017, 0124;

В) шістнадцяткові константи починаються з 0, за яким має знаходитись символ X або x, а потім одна або більше шістнадцяткових цифр (тобто цифри від 0 до 9 та латинські літери від A до F або від a до f), наприклад: 0X17, 0XA1; 0x12b.

2. Дійсні константи мають десяткову крапку або порядку (позначається e), або те і інше, наприклад: 1.2, 1e-10, -1.5e+05.

3. Символьна (інколи літерна) константи – це один символ, узятий в одинарні лапки, наприклад **`'a'`** або **`'0'`**. Така константа має ціле значення, яке збігається з кодом даного символу в таблиці ASCII. Деякі символьні константи використовуються як керуючі символи для позначення, наприклад, нового рядка – **`'\n'`** або табуляції – **`'\t'`**. Ці керуючі символи ще називаються esc-послідовностями і зображуються двома літерами, перша з яких знайомий нам слеш. Будь-який символ кодової таблиці можна в свою чергу задати у вигляді esc-послідовності з вісімковим або шістнадцятковим кодом, наприклад, **`'\0'`** – це символ з нульовим кодом (так звана null-літера), а не цифра 0; **`'\x41'`** – це те саме, що символ **`'A'`**. Символи слеш, одинарні та подвійні лапки із зрозумілих міркувань теж зображуються у вигляді esc-послідовностей : **`'\\'`** – слеш; **`'\''`**– одинарні лапки; **`'\"'`** – подвійні лапки.

4. Стрінгова (рядкова або текстова) константа – це нуль або більше символів, узятих в подвійні лапки, наприклад **`"String constant"`**, **`""`** – порожній стрінг. У внутрішньому поданні стрінгова константа займає на одну одиницю пам'яті більше, ніж кількість символів в стрінгу, бо останнім символом автоматично записується null-літера **`'\0'`**, яка завершує будь-який стрінг. Тому зрозуміло, що **`'x'`** та **`"x"`** – різні сутності.

3. Основні стандартні типи та змінні.

В С/С++ будь-яка змінна програми має бути описана (тобто має бути визначений її тип), перш ніж бути використаною. Вказуючи тип змінної, тим самим визначаються: об'єм пам'яті, виділений для збереження змінної, діапазон її можливих значень, а також допустимі операції над цією змінною. Як правило (але не обов'язково), описи (або декларації) всіх змінних поміщають на початку функції.

Декларація має вид:

```
<тип> <список змінних>;
```

або

```
<тип> < змінна> = < початкове значення>;
```

В останньому рядку змінна одержує початкове значення, відбувається її ініціалізація. Це відбувається саме на етапі компіляції програми. Основні стандартні типи мови С містяться в таблиці (у вашому конкретному компіляторі об'єм пам'яті для деяких типів може відрізнятись!):

Тип	Об'єм пам'яті (байтів)	Діапазон значень
int (цілий)	2	-32767..32768
unsigned (цілий без знаку)	2	0..65535
short (короткий цілий)	2	Аналогічний типу **int**
long (довгий цілий)	4	$-2^{31}..2^{31}-1$
float (дійсний)	4	$\pm 3.4e^{-38}..\pm 3.4e^{+38}$
double (довгий дійсний)	8	$\pm 1.7e^{-308}..\pm 1.7e^{+308}$
char (символьний)	1	-127..128
unsigned char (без знаку)	1	0..255

Зауваження.

1. Символьний тип є підмножиною цілого типу. Значенням змінної символьного типу є код відповідного символу в кодовій таблиці символів.
2. При визначенні константи (літералу) цілого типу можна приписати до неї літеру **u** (unsigned) або **l** (long).
3. Зверніть увагу на відсутність логічного типу в ANSI-стандарті мови С – для його інтерпретації використовують угоду: будь-який вираз з ненульовим значенням інтерпретується як ІСТИНА, а нульове значення знаменує ХИБНІСТЬ.

Діапазон типів мови С++ ширший – додатково до перелічених включає і такі типи:

Тип	Об'єм пам'яті (байтів)	Діапазон значень
bool (логічний)	1	0 **(false)** .. 1 **(true)**
long long int (довгий довгий цілий)	8	$-2^{63}..2^{63}-1$
long double (довгий дійсний)	10	$\pm 3.4e^{-4932}\ldots\ \pm 3.4e^{+4932}$

Приклади декларацій змінних:

```
int           i = 0, size_of_array = 10;
float         x, y, z;
char          ch = 'A';
unsigned      score = 0;
unsigned long lnum = 0L;
```

4. Вирази та основні операції.

Вираз – це допустима конструкція з лексем мови. Елементи виразу називаються операндами, а дії над ними – операціями. Операції бувають унарними, якщо виконуються над одним операндом, та бінарними, якщо мають два операнди. Слід зазначити, що вирази завжди

виконуються лише над операндами однакових типів, а в разі необхідності відбувається неявне приведення (якщо це можливо) до найбільш широкого типу. Крім того, операції об'єднуються в наступні групи операції.

4.1. Арифметичні операції.

Арифметичні операції застосовуються до операндів цілих, дійсних та символьного типів. Позначаються наступними знаками:

+ – додавання;
- – віднімання, або протилежне значення;
***** – множення:
/ – ділення;
% – взяття остачі від ділення (призначена для цілих операндів).

Тип результату визначається типом операндів, тобто при діленні 1/2 в результаті матимемо нуль, при діленні 1.0/2 – результат, рівний 0.5

Приклади.

1. **1.5*2** дає в результаті 3.0;
   15%2 дає в результаті 1;
   15/2 дає в результаті 7 (оскільки обидва операнди цілі, то і результат виразу буде цілим)
2. Нехай в програмі є декларація:

```
unsigned  j;
```

Припустимо, що змінна **j** має значення 65535 (максимальне значення типу **unsigned**). Тоді результатом додавання **j + 1** буде 0.

4.2. Операції відношення.

Бінарні операції відношення задають виконання однієї з перевірок стосовно двох операндів (однакових типів):

== – перевірка операндів на рівність;
!= – перевірка операндів на нерівність;
< – перевірка умови, що перший операнд менший за другий;
> – перевірка умови, що перший операнд більший за другий;
<= – перевірка умови, що перший операнд не більший за другий;
>= – перевірка умови, що перший операнд не менший за другий.

Всі операції відношення визначають цілий результат (в C++ - результат матиме тип **bool**), він рівний 1, якщо результатом перевірки є істина, і 0 у супротивному випадку.

Приклади.

1. **1 < 2** дає в результаті 1 (**true**);
2. **1 == 2** дає в результаті 0 (**false**);
3. **1.5 != 2** дає в результаті 1 (**true**).

4.3. Логічні операції.

Логічні операції виконуються над операндами цілих, дійсних, символьного типів. Операнд трактується як "хибність", якщо він рівний 0 і як "істина" у супротивному випадку. Логічні операції позначаються символами:

&& – логічне множення або логічне "І";
|| – логічне додавання або логічне "АБО";

! – логічне заперечення, унарна операція, яка змінює логічне значення на протилежне.

Нижче наведені таблиці результатів бінарних логічних операцій (з очевидною поправкою для С++: замість 1 – **true**, замість 0 – **false**)

&&	1	0
1	1	0
0	0	0

||	1	0
1	1	1
0	1	0

Результатом будь-якої логічної операції буде 1 в разі істини та 0 в разі хибності.

Приклади.

1. Вираз **x && y** еквівалентний **(x != 0) && (y != 0);**
2. Вираз **!(x || y)** еквівалентний **(x = 0) && (y = 0);**
3. Вираз **!x** еквівалентний **(x == 0).**

4.4. Побітові операції.

Побітові операції забезпечують виконання певних логічних операцій над бітами внутрішнього подання операндів **тільки цілих типів**. Побітовими операціями є наступні:

& – побітове «І» (кон'юнкція, множення відповідних бітів двох операндів);

| – побітове «АБО» (диз'юнкція бітів);

^ – побітове виключне «АБО»;

<< – побітовий зсув вліво лівого операнду на кількість бітів, вказану правим операндом;

>> – побітовий зсув вправо лівого операнду на кількість бітів, вказану правим операндом;

~ – побітове заперечення (унарна операція, в результаті якої кожний біт операнду інвертується, тобто змінює своє значення на протилежне).

Нижче наведені таблиці результуючих бітів при виконанні бінарних побітових операцій

&	1	0
1	1	0
0	0	0

|	1	0
1	1	1
0	1	0

^	1	0
1	0	1
0	1	0

Зауваження. При побітовому зсуві вліво вільні позиції праворуч заповнюються нулями; при побітовому зсуві вправо вільні позиції ліворуч гарантовано заповнюються нулями лише у випадку беззнакового операнда, в деяких компіляторах при зсуві вправо від'ємного операнду відбувається розмноження знакового розряду, тобто вільні позиції ліворуч заповнюються одиницями.

Крім того, очевидно, що побітовий зсув вліво на один розряд означає множення операнду на 2, а побітовий зсув вправо на один розряд означає його цілочисленне ділення на 2.

Приклади.

1. Нехай в програмі є декларація:

```
unsigned  j = 1,          /* 0000 0000 0000 0001 */
          k = 2,          /* 0000 0000 0000 0010 */
          mask = 0xF,     /* 0000 0000 0000 1111 */
```

Тоді результатом виразу **j && k** буде 1 (**true**), результатом виразу **j & k** буде 0, результатом виразу **j || k** буде 1 (**true**), результатом виразу **j | k** буде 3, результатом виразу **j << 3** буде 8, результатом виразу **k >> 1** буде 1, результатом виразу **~k** буде 65533. Вираз **k ^ mask** дасть результат **0000 0000 0000 1101,** це число 13, тобто одиничні біти маски при виконанні операції побітового виключного «АБО» інвертують відповідні біти операнду **k**, а нульові біти маски – не змінять зміст відповідних бітів числа **k**.

4.5. Операції присвоєння.

Крім операції присвоєння виду

<змінна> = <вираз>;

яка полягає в обчисленні значення виразу і присвоєння його змінній в лівій частині, є ще десять операцій присвоєння виду:

<змінна> op= <вираз>;

Тут **op** – це знак однієї з операцій: **+ - * / % << >> & | ^**. Виконується як присвоєння:

<змінна> = <змінна> op <вираз>;

Приклади.

1. Вираз **x += 2;** еквівалентний **x = x + 2;**
   Вираз **x *= 10;** еквівалентний **x = x * 10;**

2. Операції присвоєння в одній інструкції виконуються послідовно справа наліво:

```
x = y = z = 0;
```

еквівалентно

```
z = 0;
y = z;
x = y;
```

4.6. Операції інкременту та декременту.

Операція інкременту є унарною (тобто застосовується тільки до одного операнда), збільшуючи змінну одного з цілих типів на 1 і позначається **++** , а операція декременту – зменшує змінну на 1 і позначається **--**. Знак будь-якої з цих операцій може стояти перед змінною, наприклад, **++k** (префіксна форма), або після змінної, наприклад, **k++** (постфіксна форма). В першому випадку змінна відповідним чином змінюється, а потім використовується. В другому – навпаки: спочатку використовується значення змінної, а потім вона змінюється.

Приклади.

1. Нехай в програмі є декларація:

```
int  n = 10, m;
```

Тоді результатом присвоєння **m = ++n;** буде 11, а результатом присвоєння **m = n++;** буде значення **m** рівне 10, хоча в обох випадках змінна **n** матиме значення 11.

Зауваження. Операції інкременту та декременту застосовні тільки до змінних, але не до виразів, тобто вираз **(m + n)++;** є неприпустимим.

7. Умовна операція (умовний вираз).

Синтаксис умовної операції:

```
вираз1 ? вираз2 : вираз3;
```

Порядок виконання умовної операції такий: обчислюється значення **виразу1**. Якщо він є істиною (тобто не дорівнює 0), то обчислюється значення **виразу2** і саме воно є результатом

умовної операції. Якщо ж **вираз1** є хибним (тобто рівним 0), то обчислюється значення **виразу3** і саме воно буде результатом всього умовного виразу.

Приклад.

1. `/* Тут обчислюється  z = max (x, y) */`

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

Порядок виконання операцій у виразі визначається пріоритетом операцій або дужками. Ніколи не слід економити на дужках – розставити їх легше, ніж пам'ятати пріоритети операцій, які визначаються наступною таблицею (операції, які знаходяться в одному горизонтальному рядку таблиці – мають рівний пріоритет і виконуються послідовно, в тому порядку, який зазначений в правому стовпчику таблиці).

Операції	Порядок виконання
`! ~ ++ -- + - *` (унарні операції)	Справа наліво
`* / %` (операції типу множення)	Зліва направо
`+ -` (операції типу додавання)	Зліва направо
`<< >>` (операції зсувів)	Зліва направо
`< <= > >=` (операції відношення)	Зліва направо
`==  !=` (операції відношення)	Зліва направо
`&` (побітове множення)	Зліва направо
`^` (побітове виключне АБО)	Зліва направо
`|` (побітове додавання)	Зліва направо
`&&` (логічне множення)	Зліва направо
`||` (логічне додавання)	Зліва направо
`? : ;` (умовний вираз)	Справа наліво
`= += -= *= /= %= &= ^= |= <<= >>=` (операції присвоєння)	Справа наліво

Завдання для самоперевірки.

1. Нехай змінна **n** має значення 1. Визначити значення змінної **k = n+++1** та змінної **n** після обчислення **k**.
2. Нехай в програмі є декларація:

```
unsigned  j=5, k=3;
```

Визначити результати виразів: **j&&k, j&k, j|k, k<<2, j >>1, ~k** .

3. Пояснити, чому результатом виразу **1/(n + 1)** буде 0 при будь-якому натуральному значенні цілої змінної **n** та як досягти дійсного значення результату ділення.
4. З допомогою умовного виразу визначити **z = min (x, y).**

Контрольні питання.

1. Назвіть основні стандартні типи мов C/C++ .
2. Навіщо потрібно описувати змінні в програмі?
3. Назвіть групи операцій в C/C++ та пригадайте, як виконуються ці операції.
4. Що таке пріоритет операцій?

Лекція 2. Основні конструкції для керування виконанням програми мовами С/С++.

Розглянемо конструкції мови С/С++, які дозволяють забезпечити потрібну послідовність виконання операцій в програмі.

Всі керуючі конструкції можна розділити на 2 типи:

- ✓ конструкції вибору;
- ✓ цикли.

З'ясуємо перш за все, що таке інструкція (в деяких виданнях відповідний термін перекладають як «оператор» (statement)) мови С/С++? Інструкцією будемо називати вираз, що закінчується `;`. Цей знак (`;`) відокремлює одну інструкцію від іншої. Синтаксис мови, звісно, допускає запис декількох інструкцій в одному рядку програми. Ми ж намагатимемось з міркувань покращення програм для читання та сприйняття користуватись цією можливістю якнайменше. Наведемо приклади інструкцій.

1.
```
/* Одна інструкція */
z = (x > y)? x : y ;
```
2.
```
/* Дві інструкції   */
x += 1; y = 0;
```

Фігурні дужки **{ }** використовуються для об'єднання декларацій та інструкцій в складену інструкцію або, іншими словами, – блок. Наприклад, інструкції функції **main** утворюють блок – вони об'єднані фігурними дужками. До речі, змінні можна декларувати на початку будь-якого блоку (не обов'язково на початку функції), тоді вони можуть використовуватись (кажуть «бути видимими») протягом виконання цього блоку. Найпростіша інструкція мови С – порожня інструкція **;** , яка не викликає ніяких дій.

Розглянемо ***умовні конструкції*** або ***конструкції вибору*** мови С – розгалуження та перемикач.

1. Умовна конструкція `if`

Її синтаксична форма така:

```
if (вираз)
     інструкція
```

Порядок виконання цієї конструкції такий: спочатку обчислюється значення **виразу.** Якщо його значення не дорівнює 0 (тобто інтерпретується як ІСТИНА), то виконується **інструкція**. У супротивному разі (якщо значення **виразу** є нульовим) **інструкція** не виконується, виконуються наступні інструкції програми. Зрозуміло, що типом **виразу** може бути довільний скалярний тип.

Зверніть також увагу, що за синтаксисом, в умовній конструкції припустима лише одна інструкція. Якщо треба виконати декілька, їх об'єднують у блок.

Приклади.

1.
```
/* Якщо x != 0, обчислюється вираз 1.0/x (вважаємо, що змінна x
має цілий тип) */
if (x) x = 1.0/x;
```

Зауваження. Зверніть увагу, як компактно виглядає перевірка змінної **x** на нерівність 0. Як можна інакше записати еквівалентну конструкцію?

2.
```
/* Якщо значення змінної y, рівної x*x – 1, є додатним,
обчислюється квадратний корінь з y (вважаємо, що обидві змінні
мають дійсний тип) */
if ((y = x*x - 1) > 0) y = sqrt (y);
```
3.
```
/* Якщо значення змінної k парне, то виведемо відповідне
повідомлення та присвоїмо їй значення наступного парного числа
(вважаємо, що змінна k має цілий тип) */
if (!(k % 2))
{
     printf ("even number");
     k += 2;
}
```
Зауваження. Зверніть увагу на необхідність фігурних дужок в цьому прикладі.

2. Конструкція `if-else`

Її синтаксична форма виглядає так:
```
if (вираз)
     інструкція1
else
     інструкція2
```
Порядок виконання: спочатку обчислюється значення **виразу.** Якщо його значення не дорівнює 0 (тобто інтерпретується як ІСТИНА), то виконується **інструкція1**. У супротивному разі (якщо значення **виразу** є нульовим) виконується **інструкція2**.
Отже, з двох інструкцій завжди виконується одна і тільки одна. Саме тому текст програми бажано структурувати відступами.
Приклади.
1.
```
/* Що виконується в даному фрагменті програми? (вважаємо, що
обидві змінні мають символьний тип) */
     if (ch >= 'A' && ch <= 'Z')
        c = ch + 'a' – 'A';
     else
        c = ch;
```
Зауваження. Дамо відповідь на запитання, поставлене у коментарі: оскільки в кодовій таблиці символів великі та малі літери розташовані послідовно, то зсув між відповідними парами малих та великих букв один і той самий, рівний **'a' – 'A'.** Таким чином, ця конструкція розгалуження перевіряє, чи є символьна змінна **ch** великою літерою, і, якщо це так, «переводить її на нижній регістр» і присвоює одержане значення змінній **c**. Якщо ж **ch** не є великою літерою, то її значення просто присвоюється змінній **c**.
2.
```
/* Як буде виконуватись даний фрагмент програми? (вважатимемо,
що всі змінні мають один і той самий числовий тип)  */
     if (t > 0)
        if (y > x)
           z = y;
        else z = x;
```
Зауваження. Проблема в тому, що дві умовні конструкції мають одну **else**-частину, тому виникає питання, до якого саме **if** вона відноситься? Правильна відповідь така: якщо в

програмі зустрічається відокремлена **else**-частина, то вона відноситься до самого останнього, внутрішнього **if** (що, власне, підкреслено відступами). Якщо потрібний інший порядок виконання, то треба використати фігурні дужки. Отже, при **t > 0** змінній **z** буде присвоєне значення максимуму з **x** та **y**.

2.А. Конструкція if-else-if

Ця конструкція не є окремою, а являє собою просто послідовність вкладених розгалужень **if-else-if**. Часто використовується для виконання багатоступеневих перевірок.
Приклад.

```
/* Аналіз знаку арифметичної операції і виконання цієї операції
над операндами x та y. Результат записується в змінну z */
char sign;
int  x, y, z;
/* Вважатимемо, що тут всі змінні sign, x, y, z
вже проініціалізовані */
if (sign == '*')
   z = x*y;
else if (sign == '/')   /* Вважатимемо, що y не дорівнює 0.*/
        z = x/y;
      else if (sign == '+')
              z = x + y;
           else if (sign == '-')
                   z = x - y;
                else if ( sign == '%')
                  z = x%y;
                      else
                     printf ("\n Wrong operation");
```

3. Перемикач switch

Конструкція перемикача використовується для вибору одного з багатьох варіантів.
Його синтаксична форма виглядає так:

```
switch (вираз)
{
     case константний_вираз1   :  інструкція1
     case константний_вираз2   :  інструкція2
                    ⋮
     case константний_вираз_N  :  інструкціяN
     default                   :  інструкція
}
```

Відразу зауважимо, що допустимим типом **виразу** є будь-який цілий тип або символьний (в С++, очевидно допускається і логічний тип). Типи всіх **константних_виразів** мають збігатись із типом **виразу**. Порядок виконання перемикача такий: обчислюється значення **виразу**, що стоїть в дужках; потім здійснюється перехід на ту вітку **case**, **константний_вираз** якої має значення **виразу**, Якщо такої вітки немає, то відбувається

перехід на інструкцію, що відповідає частині **default**. Якщо ж немає і **default**, то відбувається вихід з усієї конструкції.

Зауваження щодо особливості виконання:

- ✓ вітки **case** та **default** можуть бути розміщені в довільному порядку;
- ✓ перехід на відповідну вітку відбувається як перехід на мітку, тобто починають виконуватись всі інструкції підряд, які є у наступних вітках. Якщо в цьому немає необхідності, використовують інструкцію **break;** , яка перериває виконання інструкцій всього блоку.

Приклад – реалізуємо попередній приклад з допомогою перемикача.

```
/* Аналіз знаку арифметичної операції і виконання її над
операндами x та y – використовуємо switch. Результат записується
в змінну z */
char sign;
int  x, y, z;
/* Вважатимемо, що тут всі змінні sign, x, y, z
вже проініціалізовані */
switch (sign)
{
     case '*'   :   z = x*y;
                break;
     case '/'   :   z = x/y;
                break;
     case '+'   :   z = x + y;
                break;
     case '-'   :   z = x - y;
                break;
     case '%'   :   z = x%y;
                break;
     default    printf ("\n Wrong operation ");
                break;
}
```

Зауваження. Остання інструкція **break;** не є необхідною, а використана на випадок продовження блоку даної конструкції вибору.

Розглянемо далі ***цикли*** в C/C++. Цикли дозволяють реалізувати повторне виконання певних обчислень, наприклад, організувати обчислення сум, добутків, тощо.
Три види циклів, які розглядаються далі, відрізняються механізмами керування їх виконанням.

4. Цикл for

Цей цикл зручно використовувати, коли визначені точна кількість кроків повторення Його синтаксична форма така:

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

Інструкція називається «тілом циклу». Якщо необхідно виконувати в тілі циклу більше однієї інструкції, то всі їх необхідно об'єднати у блок. Взагалі кажучи, кожний з **виразів** циклу for може використовуватись довільно, але як правило, **вираз1** пов'язаний з

ініціалізацією циклу; **вираз2** – це контроль продовження виконання циклу, **вираз3** – реалізує ітерацію циклу.

Кожний з **виразів** (або навіть всі) можуть бути пропущені, але розділові знаки **;** обов'язкові.

Порядок виконання циклу:

1. Обчислюється **вираз1** (якщо він присутній);
2. Обчислюється **вираз2** (якщо він присутній). Якщо значення **виразу2** є ХИБНІСТЬ (тобто 0), то цикл завершено. В супротивному разі, або якщо **вираз2** відсутній, цикл продовжується.
3. Виконується тіло циклу, тобто **інструкція**.
4. Обчислюється **вираз3** (якщо він присутній).
5. Здійснюється перехід до пункту 2.

Зауваження 1. Зверніть увагу, що послідовно та повторно виконуються кроки 2-3-4, крок 1, тобто обчислення **виразу1** відбувається лише раз на початку циклу.

Зауваження 2. Якщо в будь-якому місті тіла циклу **for** зустрінеться інструкція **break**, то відбудеться вихід з циклу, якщо ж зустрінеться інструкція **continue**, то відбудеться перехід до пункту 4, тобто пропускається частина тіла циклу після інструкції **continue**.

Приклади

1.
```
/* Нескінченний цикл */
for ( ; ; )
{
     /* інструкції циклу for */
}
```

Серед інструкцій цього циклу обов'язково має бути інструкція, яка його перериває, наприклад, **break;**.

2. /* Обчислення суми доданків виду $\frac{1}{(i+1)}$ від $i = nn$ до $i = nk$ */

```
#include <stdio.h>
main ()
{
     int     i, nn, nk;
     float   sum = 0.0;     // змінна для суми
     printf ("\n Input: nn = ");
     scanf  ("%i",&nn );    // вводимо значення змінної nn
     printf ("\n Input: nk = ");
     scanf  ("%i",&nk );    // вводимо значення змінної nk
        for (i = nn; i <= nk; i++)
        sum += 1.0/(i + 1);
     // виводимо обчислене значення суми sum
     printf ("\n Output: sum = %f", sum);
     return 0;
}
```

Зауваження. Зверніть увагу – змінна циклу **i** пробігає значення від **nn** до **nk**, збільшуючись на кожному кроці на 1.

3.
```
/* Підрахунок кількості ненульових бітів у внутрішньому
представленні цілого числа x */
unsigned  x;
int       score;
```

```
/* Тут число x має бути проініціалізованим */
   for (score = 0; x != 0; x >>= 1)
      if (x & 01) score++;
```

Зауваження. Зверніть увагу – у **виразі3** відбувається зсув змінної **x** на одну позицію праворуч. Як ми пам'ятаємо, для типу **unsigned** позиції, що вивільняються зліва заповнюються нулями, отже, коли всі значущі розряди будуть перевірені, змінна **x** стане рівною 0, і цикл завершиться. Бітове множення на вісімкову константу **01** дає результат, рівний 1 (ІСТИНА), якщо останній біт в числі **x** ненульовий. Таким чином, після завершення циклу змінна **score** міститиме кількість ненульових бітів цілого числа **x**

4.
```
/* Ця функція перезаписує стрінг в зворотному порядку */
#include <cstring>
void  reverse (char s [ ])
{
   int i, j, c;
     for (i = 0, j = strlen (s) - 1;  i < j;  i++, j++)
     {
     c = s [i];
     s [i] = s [j];
     s [j] = c;
     }
}
```

Зауваження. Зверніть увагу: ми дещо забігаємо наперед – адже ми ще не обговорювали правила написання методів і використання масивів (текстовий стрінг **s** – це символьний масив), проте приклад дуже корисний і змістовний.

В заголовку циклу **for** використовується операція слідування (послідовного виконання), яка позначається комою. Вона дозволяє об'єднувати декілька виразів в один. Таким чином в ініціалізації циклу обидві змінні циклу **i** та **j**, які в циклі «рухаються назустріч одна одній», одержують початкові значення: перша вказує на початковий символ стрінгу, друга – на останній. Так само у **виразі3** змінюються вони обидві: одна збільшується, друга – зменшується. Ну, в цикл триває допоки ці змінні «не зустрінуться».

5. Цикл while (цикл з передумовою).

В даному виді циклу перед кожним кроком виконання перевіряється умова продовження циклу. Його синтаксис такий:

```
while (вираз)
      інструкція
```

Порядок виконання циклу:

1. обчислюється значення **виразу**;
2. якщо значенням **виразу** є ХИБНІСТЬ (тобто 0), то цикл завершено. В супротивному разі, тобто коли значення виразу відмінне від 0 (ІСТИНА) виконується тіло циклу – **інструкція.**
3. Здійснюється перехід до пункту 1.

Як правило, тіло цього циклу – це блок інструкцій, серед яких обов'язково має бути інструкція, що впливає на значення **виразу**, який перевіряється перед кожним виконанням тіла циклу.

Приклади

1.
```
/* Аналог циклу for (i = 1; i <= n; i++) інструкція */
int i = 1;
while (i <= n)
{
     інструкція
     i++;
}
```
2.
```
/* Копіювання символів вводу на екран до першої крапки */
#include <stdio.h>
main ()
{
     int     ch;
     printf ("\n Input text with . at the end \n");
        while ((ch = getchar ()) != '.')
        putchar (ch);  // виводимо на екран черговий символ
     putchar (ch);    // після закінчення циклу виводимо крапку
     return 0;
}
```

Зауваження. Зверніть увагу: в умові циклу **while** одночасно зчитується черговий символ (його повертає функція **getchar ()**) і відбувається перевірка, чи не рівний він символу **.** (крапка), який має бути останнім символом вводу.

6. Цикл do-while (цикл з післяумовою).

В даному циклі умова продовження циклу перевіряється після виконання тіла циклу. Таким чином цей цикл виконується завжди принаймні один раз. Його синтаксис такий:

```
do
     інструкція
while (вираз);
```

Порядок виконання циклу:

1. виконується тіло циклу – **інструкція**;
2. обчислюється значення **виразу**;
3. якщо значенням **виразу** є ХИБНІСТЬ (тобто 0), то цикл завершено. В супротивному разі, тобто коли значення **виразу** відмінне від 0 (ІСТИНА), відбувається перехід до пункту 1.

Приклади

1.
```
/* Обчислення n-го степеня числа x */
int i = 1;
float p = 1.0;
/* Тут мають бути визначені значення
цілої змінної n та дійсної x */
   do
   {
     p*=x;
     i++;
   }
   while (i <= n);
```
2.
```
/* Реалізація запиту на повторне виконання програми,
```

```
   повтор відбувається при натисканні 1 */

#include <stdio.h>
main ()
{
     int     answ;
     do
     {
  /*****  Тут виконуються оператори програми *****/
          printf ("\n If you wan\'t to repeat, press 1 \n");
          scanf ("%i",&answ);
     }
     while (answ == 1);
     return 0;
}
```

2.А)

```
/* Обчислення суми з прикладу 2 із запитомна повторне виконання.
   Версія для потоків вводу-виводу */
#include <iostream>
using namespace std;
main ()
{
   int     i, nn, nk, answ;
   float   sum;
        do
        {
           cout << "\n Input: nn = ";
           cin  >> nn;
           cout << "\n Input: nk = ";
           cin  >> nk;
           sum = 0.0;
              for (i = nn; i <= nk; i++)
              sum += 1.0/(i+1);
           cout << "\n Output: sum = " << sum;
           cout << "\n If you wan\'t to repeat, press 1 \n";
           cin >> answ;
           cout << answ;
        }
      while (answ == 1);
   return 0;
   }
```

Контрольні питання.

1. З якими умовними конструкціями мов С/С++ ви познайомились, в чому різниця між ними?
2. Які види циклів є в С/С++ ?
3. Як ви гадаєте, у яких випадках який саме з видів циклів вигідно використовувати?

Завдання для самоперевірки.

1. Написати програму, яка перевіряє, чи є заданий користувачем рік високосним і друкує результат перевірки. Програма має контролювати правильність вхідних даних (тобто не допускати, наприклад, від'ємний рік).

2. Написати програму, яка по введеному користувачем номеру місяця виводить назву пори року. Програма має контролювати правильність вхідних даних (тобто не допускати, наприклад, значення місяця за межами значень від 1 до 12).
3. Написати програму, яка по введеному користувачем номеру дня тижня виводить назву дня тижня. Програма має контролювати правильність вхідних даних (тобто не допускати, наприклад, значення дня за межами значень від 1 до 7).
4. Написати програму, яка обчислює суму парних натуральних чисел від 2 до 100.
5. Написати програму, яка обчислює суму непарних натуральних чисел від 1 до 99.
6. Написати програму, яка обчислює кількість символів у заданому стрінгу. Введення стрінгу має завершуватись крапкою.

Лекція 3. Вказівники. Динамічні змінні. Масиви.

1. Поговоримо про пам'ять – адресація та вказівними.

Вказівник (або покажчик, інколи – посилання) – це змінна, яка може містити адресу деякого іншого об'єкта. Ним може бути в найпростішому випадку інша змінна або масив, структура, функція тощо.

Вказівники широко використовуються в C та C++ не лише тому, що вони дозволяють писати більш короткі та ефективні програми, а й в основному тому, що без них часто просто неможливо обійтись.

Перш ніж говорити про адресацію та вказівники, спробуємо уявити, що собою являє пам'ять типової обчислювальної машини. Пам'ять – це послідовність перенумерованих комірок, а оскільки їх досить багато, то для їх нумерації використовують 16-кову систему числення. До більшості з цих перенумерованих комірок, які ми називаємо байтами пам'яті, можна звернутись безпосередньо з програми, вказавши її номер, який і є її адресою.

Отже, вказівник – це 2 (інколи 4) байти в пам'яті, в яких записана адреса (номер першого байту місця розташування) деякого іншого об'єкту.

Коли ми звертаємось до будь-якої змінної, вказуючи її ідентифікатор (ім'я), ми просто перекладаємо на компілятор всю роботу по, по-перше, розшуку в пам'яті місця знаходження змінної (компілятор заводить щось на зразок адресної книги, в якій зберігаються адреси розташувань всіх наявних у програмі змінних), а по-друге, по вибору з цього місця її значення. Такі змінні, пам'ять для яких виділяється автоматично і до яких звертання відбувається просто за ідентифікатором, називаються статичними.

Важливо зрозуміти різницю між самою змінною і її адресою, тобто значенням вказівника на неї. Ця різниця приблизно така сама, як і між будинком з усім його вмістом та номером цього будинку. (Правда, у змінних адреси індивідуальні).

Нехай в програмі визначена деяка ціла змінна:

```
int  ival;
```

Для того, щоб визначити змінну-вказівник, яка може посилатись (іншими словами – містити адресу) на довільну змінну цілого ж типу, необхідно написати:

```
int *pi;
```

Тут **pi** – це ідентифікатор змінної вказівника на тип, а знак-зірочка – свідчення того, що ця змінна є саме вказівником. Тоді інструкція **pi = &ival;** присвоює адресу першого байту з тих двох, в яких розміщена змінна **ival**, змінній **pi.** Отже унарна операція **&** (на відміну від однойменної бінарної операції, що задає кон'юнкцію бітів) є операцією взяття адреси, тобто визначає адресу об'єкта, який розташований в пам'яті. Вона може бути

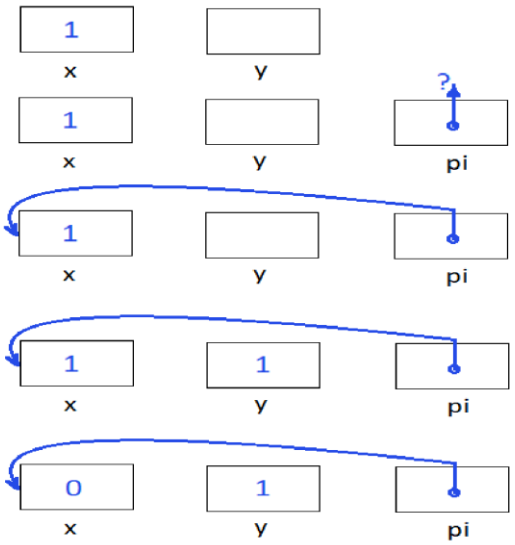
застосована лише до об'єктів, які знаходяться в оперативній пам'яті, а отже її операндом не може бути ні вираз, ні константа, ні регістрова змінна (про них дещо пізніше). Протилежну дію (тобто перехід від адреси об'єкту до його значення) виконує унарна операція ***** (зірочка). Вона називається розкриттям посилання, або розадресацією (розіменуванням) вказівника. Таким чином, ***pi** є значенням того об'єкта, на який посилається вказівник. В данному прикладі ***pi** – це сама змінна **ival.** (У декларації **int *pi;** таким чином, заявлено, що ***pi** має цілий тип)

Зауваження. Тип вказівника при його визначенні прив'язується до типа об'єкта, на який він може посилатись. Це й зрозуміло, адже об'єкти різних типів займають різні за об'ємом області пам'яті.

Приклади.

1.
```
/* Взяття адреси та розадресація */
int  x = 1, y;
int  *pi; /* Тут визначається змінна-вказівник, але ніякої
     адреси вона поки що не містить */
pi   = &x; /* Вказівник pi ініціалізується адресою змінної x:
тепер x та *pi – одна й та сама змінна  */
y    = *pi;/* Змінній y присвоєно значення змінної x, тобто 1*/
*pi = 0;  /* Змінній x присвоєно значення 0 через вказівник */
```

На малюнку нижче відображено виконання послідовних інструкцій цього прикладу



2.
```
/* Продовження прикладу 1. Операції зі змінними, що адресуються
вказівниками */
*pi = *pi + 10;      // Змінна x збільшена на 10
*pi += 1;            // Змінна x тепер рівна 11
```

Зауваження. Остання інструкція еквівалентна одній з інструкцій **++*pi;** або **(*pi)++;**

У постфіксному інкременті дужки обов'язкові, бо хоч операції **++** та ***** мають однаковий пріоритет, але виконуються справа наліво, тому ***pi++;** означає спочатку збільшення на одиницю значення вказівника, а вже потім визначення значення об'єкта, що міститься за новою адресою.

2. Вказівники та масиви.

В мовах C/C++ існує самий безпосередній зв'язок між вказівниками та масивами змінних. ***Масив*** – це структурований тип даних, який містить фіксовану кількість елементів одного й того самого типу. Доступ до кожного елементу масиву відбувається за його номером в масиві, або як кажуть – за індексом. Індексуються масиви в C/C++, починаючи з 0. Синтаксис визначення статичного масиву має вид:

```
тип_елементів   ідентифікатор_масиву   [кількість_елементів];
```

Приклади.

```
int   vector [10];  // vector – масив із 10 цілих елементів
float mas [100];    // mas – масив із 100 дійсних елементів
char  line [60];    // line – масив із 60 символьних елементів
```

Елементи масиву, таким чином, нумеруються від 0 до **кількість_елементів - 1**, тобто елементами першого масиву є: **vector [0], vector [1],..., vector [9].**

При визначенні масиву компілятор резервує

sizeof(тип_елементів)*кількість_елементів

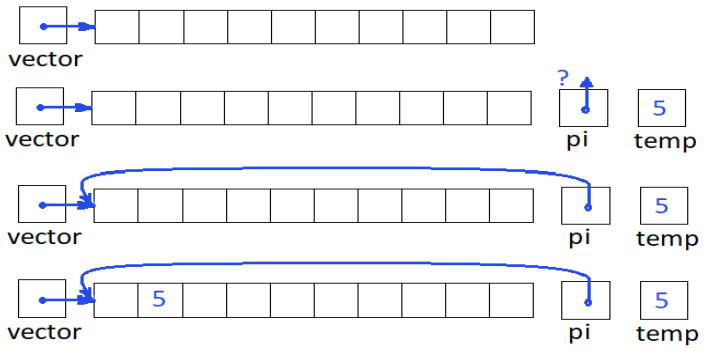
байтів пам'яті для збереження його елементів (операція **sizeof(тип)** повертає об'єм пам'яті в байтах, яку займає об'єкт вказаного у дужках типу). Таким чином, перший з масивів займатиме **sizeof(int)*10** байтів.

Важливо запам'ятати, що ***ідентифікатор статичного масиву*** є константою-вказівником, який вказує на адресу початку масиву, тобто на його нульовий елемент.

Приклади.

```
int   vector [10]; // vector – вказівник на статичний масив
int   *pi, temp = 5;
pi   = &vector [0];/* Вказівник pi вказує на нульовий елемент
                      масиву vector */
pi   = vector;    /* Ця інструкція еквівалентна попередній */
*(pi + 1) = temp ;  /* Наступному після нульового, тобто першому
             елементу масиву присвоєно значення змінної temp */
```

На малюнку нижче зображено виконання інструкцій цього прикладу:



Зауваження. За означенням, якщо **pi** – це вказівник на масив, то **pi + k** вказує на його **k**-ий елемент.

Таким чином, наступні вирази є еквівалентними:

&vector [k] та **vector + k** – це один і той самий вказівник;

***(vector + k)** та **vector[k]** – це один і той самий елемент масиву.

Адресна арифметика.

Вказівники, що вказують ***на один і той самий тип***, можна присвоювати один одному, перевіряти на рівність та нерівність.

Вказівникам будь-якого типу можна присвоювати значення **NULL** - це нульовий вказівник, який нікуди не посилається. Можна порівнювати з **NULL** на рівність та нерівність будь-який вказівник.

Якщо **p1** та **p2** – вказівники ***на один і той самий масив,*** то додатково з ними допустимі такі операції:

p1 < p2 перевірка: «**p1** вказує на елемент масиву з меншим індексом ніж той, на який вказує **p2**»;

p1 <= p2 перевірка: «**p1** вказує на елемент масиву з індексом не більшим ніж той, на який вказує **p2**»;

p1 > p2 перевірка: «**p1** вказує на елемент масиву з більшим індексом ніж той, на який вказує **p2**»;

p1 >= p2 перевірка: «**p1** вказує на елемент масиву з індексом не меншим ніж той, на який вказує **p2**»;

p2 – p1 кількість елементів масиву між елементами, що адресуються вказівниками **p2** та **p1**;

p1 + k вказівник на **k**–ий елемент масиву, рахуючи вперед від елементу, що адресується **p1**.

p1 - k вказівник на **k**–ий елемент масиву, рахуючи назад від елементу, що адресується **p1**.

Важливо пам’ятати, що при виконання всіх цих операцій вихід значення вказівника за межі масиву не контролюється.

Початкові значення елементам масиву можна задавати в момент його визначення, просто перелічивши їх в фігурних дужках. Наприклад:

```
int     masA [4] = {0, 1, 2, 3};
float   masF [ ] = {0.5, 1.5, 2.3};
char    text [ ] = { 'H','e','l','l','0','\0'};
```

Кількість елементів в другому масиві компілятор визначить автоматично рівним 3. Кількість елементів в третьому масиві буде 6. Останнім елементом в символьному масиві, який задається поелементно, треба вказати **null**-літеру.

3. Символьні або текстові масиви, або рядки, або стрінги.

Будь-яка послідовність символів, що завершується **null**-літерою, – це стрінг. Один із способів визначити стрінг – написати щось на зразок третього рядка попереднього прикладу. Можливі і такі визначення:

```
char  text[13]  = "Hello, Jack!";
```

або

```
char* ptext = "Hello, Mary!";
```

Останній приклад істотно відрізняється від попередніх, бо, по-перше, таке визначення можливе лише для символьних масивів, а по-друге, **ptext** є вказівником, який вказує на стрінгову константу, але сам цей вказівник не є константою на відміну від імен масивів в попередніх рядках. Тобто значення самого вказівника можна змінити і він буде посилатись на

інший символ або символьний масив, тому присвоєння **ptext = text;** є допустимим, а навпаки: **text = ptext;** – ні.

Приклади (ці приклади надихані книжкою [2]).

1.
```
/* Функція повертає довжину стрінга */
#include <stdio.h>
int strlen (char* s)
{
   char* p = s;   /* Вказівник p ініціалізуємо значенням s */
      while (*p != '\0')    /* Поки черговий символ не є
                               null-літерою */
      p++;   /* просуваємо вказівник на наступний символ */
   return p - s;  /* Повертаємо різницю між останнім
                     та першим вказівниками – тобто кількість
                     символів у стрінгу*/
}
main ()
{
   char* ptext = "1234567890";;
   int len;
   len = strlen (ptext);    // викликаємо функцію strlen
   printf ("\n %i", len);   // друкуємо результат функції
   getchar ();              // просто затримка – очікування вводу
   return 0;
}
```

Зауваження. Чи враховується тут **null**-літера? Виконайте цей приклад для перевірки.

2.
```
   // Функція strcpy копіює стрінг t в s
   // Варіант 1 – використовуємо індексовані масиви
void strcpy (char s[], char t[])
{
   int i = 0;
// Присвоюємо черговий символ поки не дійдемо до null-літери
      while ((s[i] = t[i]) != '\0')
      i++;        // просуваємо індекс вперед

}
  // Варіант 2 – використовуємо властивості вказівників
void strcpy (char* s, char* t)
{
      while ((*s = *t) != '\0') // поки не переписали null-літеру
      {
        s++;   // просуваємо вперед вказівник у першому масиві
        t++;   // просуваємо вперед вказівник у другому масиві
      };
}
  // Варіант 3 – компактний
void strcpy (char* s, char* t)
{
      while ((*s++ = *t++) != '\0') /* постфіксний інкремент
```

```
        забезпечує просування вказівників після їх розадресації */
            ;  // тіло циклу порожнє, всі дії в заголовку!
}
   // Варіант 4 – оскільки результатом присвоєння null-літери
   // є значення 0, тобто ХИБНІСТЬ, то порівняння з
   // null-літерою є зайвим!
void strcpy (char* s, char* t)
{
      while (*s++ = *t++)
        ;
}
```

Зауваження. Порівняйте різні варіанти цієї функції. Який з них, на вашу думку є самим зрозумілим для вас? А який є самим коротким? Наша мета – не тільки створювати свої програми, але й «як ноти» читати програми інших програмістів задля покращення власних навичок.

4. Динамічні масиви. Розподіл пам'яті.

Визначивши вказівник на деякий тип, можна зробити цей вказівник посиланням на так звану динамічну змінну або динамічний масив. Своєю назвою вони завдячують тому факту, що ми безпосередньо можемо створювати або знищувати такі змінні по мірі потреби на відміну від статичних масивів, наприклад. Динамічні змінні розташовуються у так званій «вільній області пам'яті», яка зветься Heap – купа. До динамічних змінних не можна звернутись безпосередньо так, як ми звертаємось до звичайних змінних – за ідентифікатором. Звертання до них завжди відбувається непрямим чином, через розкриття вказівника. Динамічні змінні можна створювати в Heap та знищувати в ході виконання програми, виділяючи під них пам'ять або звільняючи її, в разі потреби. Таку можливість забезпечують стандартні функції мов C/C++ динамічного розподілу та звільнення пам'яті та, додатково, операції **new** та **delete** мови C++. Для їх використання функцій динамічного розподілу пам'яті необхідно підключити файл заголовків **cstdlib** або **stdlib.h**.

Ось декларації цих функцій:

void* malloc (n);

– повертає вказівник на **n** байтів виділеної пам'яті або **NULL**, якщо виділити пам'ять неможливо;

void* calloc (n, size);

– повертає вказівник на область пам'яті, достатньої для збереження **n** елементів масиву розміру **size** кожний, або **NULL**, якщо виділити пам'ять неможливо;

void* realloc (void* p, size_t size);

– змінює розмір пам'яті, на яку вказує вказівник **p**, на нову область пам'яті, розміром **size** байтів та повертає вказівник на нову область, або вказівник **NULL**, якщо зміна неможлива. Для частини пам'яті розміру, рівному найменшому із старого та нового значень, зміст не зміниться.

void free (p); – звільняє область пам'яті, що адресується вказівником **p**. Значення вказівника **p** мало бути одержаним від функцій **malloc** або **calloc**.

Службове слово **void** означає порожній, невизначений, в даному разі – вказівник на невизначений тип, який треба обов'язково привести до відповідного типу.

Приклад.

```
#include <stdlib.h>
```

```
main()
{
   int*   pi;      // Визначаємо вказівник на тип int
   float* pmas;    // Визначаємо вказівник на тип float
   pi = (int*) malloc (sizeof(int));  /* Розподіляємо пам'ять під
   динамічну змінну типу int, на яку буде посилатись вказівник pi
   */
   pmas = (float*) calloc (10, sizeof(float));   /* Розподіляємо
   пам'ять під 10 елементів масиву типу float, на початок масиву
   буде посилатись вказівник pmas */
   *pi = 100;      /* Ініціалізуємо значенням 100 динамічну змінну,
                   на яку посилається pi */
   for (int i = 0; i < 10; i++)
       *(pmas + i) = 0.5*i*i; /* Ініціалізуємо і-ий елемент масиву,
            через вказівник pmas */
   free (pi);      /*  звільняємо  пам'ять,  виділену  динамічній
                   змінній  */
   free (pmas);    /* звільняємо пам'ять, виділену під динамічний
             масив  */
   return 0;
 }
```

Зауваження. Зверніть увагу на те, як обробляється результати функцій **malloc** та **calloc** . Операція **(тип)** називається ***приведенням до даного типу***, а оскільки вказані функції повертають вказівник та тип **void**, то він має бути приведений до типу відповідного вказівника.

Тепер повторимо цей приклад, використовуючи можливості операцій динамічного розподілу пам'яті мови C++.

```
main()
{
   int*   pi;      // Визначаємо вказівник на тип int
   float* pmas;    // Визначаємо вказівник на тип float
   pi = new int;  /* Розподіляємо пам'ять під динамічну змінну
              типу int, на яку буде посилатись вказівник pi */
   pmas = new float [10];  /* Розподіляємо пам'ять під 10 елементів
   масиву типу float, на початок якого буде посилатись вказівник
   pmas */
   *pi = 100;      /* Ініціалізуємо значенням 100 динамічну змінну,
                   на яку посилається pi */
    for (int i = 0; i < 10; i++)
        *(pmas + i) = 0.5*i*i;   /*  Ініціалізуємо  і-ий  елемент
        масиву, через вказівник pmas */
   delete pi;        /* звільняємо пам'ять, виділену динамічній
                   змінній  */
   delete [] pmas;   /* звільняємо пам'ять, виділену під динамічний
                        масив  */
   return 0;
}
```

Зауваження. Зверніть увагу на різницю у створенні динамічної змінної та динамічного масиву. Зауважте також, що при звільненні пам'яті, виділеної під масив, після службового слова **delete** необхідні квадратні дужки – символ масиву.

5. Багатовимірні масиви.

С та С++ дають можливість задавати масиви розмірності більшої за 1. Багатовимірні масиви розглянемо на прикладі двовимірного масиву, користуючись аналогією із алгебри – прямокутною матрицею.

Синтаксис визначення двовимірного масиву:

```
<тип_елементів> <ідент-р_масиву> [к-ть_рядків] [к-ть_стовпчиків];
```

Важливо зауважити, що розмірності статичного масиву мають бути задані або цілими беззнаковими літералами, або іменованими константами, визначеними наприклад директивами препроцесора **define**. І хоча ми інтерпретуємо багатовимірні масиви як прямокутні таблиці і говоримо про «рядки» та «стовпчики», в пам'яті елементи такого масиву розташовані послідовно, таким чином, що найшвидше змінюється останній індекс, тобто у випадку двовимірного масиву – «по рядках». Якщо продекларований, наприклад, такий двовимірний масив

```
float  A [2][3];
```

то він складається і з такої послідовності елементів: **A [0][0], A [0][1], A [0][2], A [1][0], A [1][1], A [1][2]**.

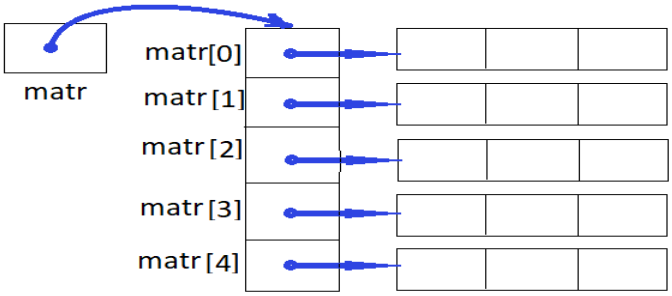
Слід також зауважити, що двовимірний масив в С/С++ насправді є масивом масивів – масивом вказівників, які вказують на масив елементів відповідного рядка.

Приклади.

1.
```
// Статичний двовимірний масив з 5*3 елементів
#include <stdio.h>
#define N_ROW 5   // тепер ідентифікатор N_ROW має значення 5
#define N_COL 3   // тепер ідентифікатор N_COL має значення 3
main()
{
   int   matr [N_ROW][N_COL]; /* Визначаємо статичний масив з 5
                                 рядків та 3 стовпчиків */
   /* Задаємо елементи масиву matr – щоб перебрати всі елементи
    потрібні 2 цикли по двом індексам */
      for (int i = 0; i < N_ROW; i++)
         for (int j = 0; j < N_COL; j++)
             matr [i][j] = (i + 1)*(j + 1);
 /* Виводимо елементи масиву matr у вигляді прямокутної таблиці */
      for (int i = 0; i < N_ROW; i++)
      {
         // Внутрішній цикл виводить всі елементи і-го рядка
         // Табулятор \t забезпечує відступи між елементами рядка
         for (int j = 0; j < N_COL; j++)
             printf ("%i\t", matr [i][j]);
         printf ("\n");  // Переводимо курсор на новий рядок
      }
   return 0;
}
```

Розмірності статичного масиву визначаються в момент створення, і не можуть бути змінені. Зрозуміло, що можна визначити досить великий масив «із запасом», а реально використовувати змінні кількості рядків та стовпчиків, які проте не перевищують реальні розмірності масиву. Значно більш гнучкою альтернативою є динамічні багатовимірні масиви. З таким двовимірним динамічним масивом можна працювати двома способами – реалізуючи його як масив вказівників на масиви рядків або ж як одновимірний масив, в якому всі елементи двовимірного масиву розташовані послідовно «ланцюжком», складеним із рядків. Розглянемо на прикладах обидві можливості.

На малюнку нижче зображено (умовно) розташуваня в пам'яті елементів статичного двовимірного масиву **`int matr [N_ROW][N_COL]`**:



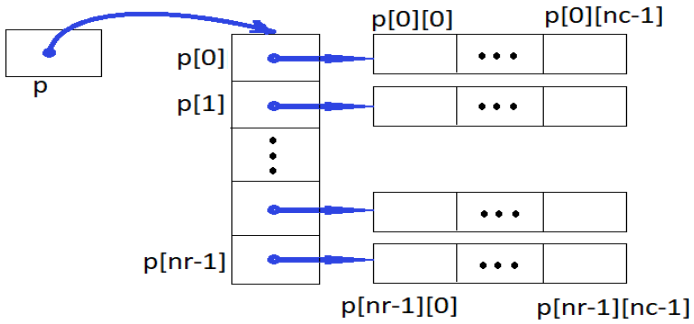
Приклади.

2.
```
/* Динамічний двовимірний масив із змінними розмірностями,
   які вводяться з клавіатури  */
#include <iostream>
#include <cstdlib>
using namespace std;
main()
{
     int nr, nc;     // Визначаємо змінні, що задають розмірності
     do              // Вводимо розмірність nr, контролюючи введення
     {
          cout << "Input numbers of rows in your array: nr = ";
          cin >> nr;
     }
     while (nr <= 0);
     do              // Вводимо розмірність nc, контролюючи введення
     {
          cout << "Input numbers of columns in your array: nc = ";
          cin >> nc;
     }
     while (nc <= 0);
     int ** p = new int * [nr];  /* Визначаємо вказівник на масив
                                    вказівників (тому дві зірочки!) та
                                    створюємо цей масив  вказівників */
     for (int i = 0; i < nr; i++)
         *(p + i) = new int [nc]; /* Створюємо масив елементів
                                     i-го рядка*/
               /* Задаємо випадкові значення із діапазону [0..20]
               для елементів двовимірного масиву */
     for (int i = 0; i < nr; i++)
```

```
        for (int j = 0; j < nc; j++)
               *(*(p + i) + j) = rand () % 21; /* Функція rand ()
               повертає випадкове число із діапазона [0..RAND_MAX],
               де  RAND_MAX – системна константа
               (операція % 21 дає випадкові остачі від 0 до 20 */
    for (int i = 0; i < nr; i++)
    {
         for (int j = 0; j < nc; j++)
               cout << p [i][j] << "\t";  /* Виводимо елементи
                                             і-рядка з табуляцією */
         cout << endl;  // Переводимо курсор на новий рядок
     }
    for (int i = 0; i < n_r; i++)
       delete [] p[i]; //звільняємо пам'ять, виділену під і-ий рядок
    delete [] p;   // звільняємо пам'ять, виділену під масив
                   // вказівників на рядки
    return 0;
}
```

Зауваження. Зверніть увагу: ***(*(p + i) + j)** та **p [i][j]** – еквівалентні звертання до елементу масиву з індексами **i** та **j**. Ще один варіант: ***(p [i] + j)**. Цикли **do-while** забезпечують контроль правильності введення розмірностей – цикл триватиме, допоки не буде введене додатне значення.

На малюнку нижче зображено розташування у вільній області пам'яті (Heap) елементів динамічного двовимірного масиву, який був використаний в цьому прикладі.



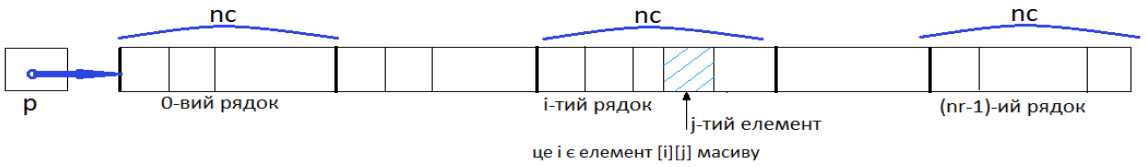
Ще один спосіб використання двовимірного масиву – це можливість інтерпретувати його як динамічний одновимірний масив, в якому елементи розташованні «ланцюжком», послідовними рядками, а місце знаходження в ланцюжку елементу **i** -го рядка та **j** -го стовпчика відповдно перерахувається.

Приклади.

3.
```
/* Динамічний двовимірний масив (ланцюжок) із змінними
   розмірностями, які вводяться з клавіатури  */
#include <iostream>
#include <cstdlib>
using namespace std;
main()
{
    int nr, nc;     // Визначаємо змінні, що задають розмірності
    do              // Вводимо розмірність nr, контролюючи введення
```

```
    {
        cout << "Input numbers of rows in your array: nr = ";
        cin >> nr;
    }
    while (nr <= 0); // Вводимо розмірність nc, контролюючи введення
    do
    {
        cout << "Input numbers of columns in your array: nc = ";
        cin >> nc;
    }
    while (nc <= 0);
    int * p = new int [nr*nc]; /* Створюємо одновимірний
                    масив-ланцюжок, що містить nr*nc елементів */
                /* Задаємо випадкові значення із діапазону [0..20]
                для елементів двовимірного масиву */
       for (int i = 0; i < nr; i++)
          for (int j = 0; j < nc; j++)
              *(p + i*nc + j) = rand () % 21; // присвоєння значення
                                              // елементу [i][j]
       for (int i = 0; i < nr; i++)
       {            // Виводимо значення елементів у вигляді таблиці
              for (int j = 0; j < nc; j++)
                   cout << p [i*nc + j] << "\t";
          cout << endl;
       }
    delete p;       // Звільняємо пам'ять, виділену під масив
return 0;
}
```

Зауваження. Зверніть увагу: **`*(p + i*nc + j)`** та **`p [i*nc + j]`** – це і є елемент двовимірного масиву з індексами **`i`** та **`j`**. Адже, щоб дістатись у ланцюжку елементів до **`i`**–го рядка, треба зсунутись від початку на **`i*nc`** елементів, а потім в цьому рядку просунутись ще на **`j`** елементів, щоб дістати вказівник на елемент з індексами **`i`** та **`j`**:



Контрольні питання.

1. Що таке вказівник, як він може бути ініціалізований?
2. Якими є індекси останнього елементу масиву, визначеного як **`int mas [4][3]`**?
3. Скажіть, що визначено в наступних деклараціях:
   ```
   char*  pline [5];
   char  (*p) [5];
   ```

Завдання для самоперевірки.

1. Написати програму, в якій визначається масив A з 10 елементів, значення яких задаються формулою A [i] = i*(i + 1), та обчислюється сума цих елементів.

2. Написати програму, в якій визначається масив B з 12 елементів, значення яких задаються формулою B [i] = 2*i – i*i, та обчислюється сума квадратів цих елементів.
3. Написати програму, в якій визначається масив C з 15 елементів, значення яких задаються формулою C [i] = 3*i*i – 2*i, та обчислюється сума кубів цих елементів.
4. Написати програму, в якій визначається двовимірний масив A з 5 на 3 елементів, значення яких задаються формулою A [i][j] = i*(j + 1) – j*j, та обчислюється максимум цих елементів.
5. Написати програму, в якій визначається двовимірний масив B з 3 на 5 елементів, які задаються формулою B [i][j] = (i + 1)*(j – 1) – 2*j*j, та обчислюється мінімум цих елементів.

Лекція 4. Введення та виведення інформації в C та C++.

Для введення і виведення інформації в C/C++ (в C++ є додаткові можливості, які обговоримо далі) використовуються функції та макроси, які знаходяться в файлі **stdio.h** (або **cstdio**), який необхідно підключити директивою препроцесора **include.**

Ці функції співпрацюють із так званими стандартними файлами: вводу (**stdin**) – позв'язаний з клавіатурою, виводу (**stdout**) та помилок (**stderr**) – обидва останні позв'язані з дисплеєм.

Найпростіший механізм введення та виведення здійснюють такі функції:

int getchar (void); – повертає черговий символ (із **stdin**) або стандартну константу **EOF** (**end-of-file**), якщо зустрівся кінець файлу введення;

int putchar (int ch); – відправляє символ **ch** у стандартний вивід (в **stdout**). Як результат повертає відправлений символ або **EOF** в разі помилки.

Приклади.

1.
```
// Змінна ch міститиме черговий символ, введений з клавіатури
#include <stdio.h>
main()
{
     int  ch;
     ch = getchar ();
}
```

2.
```
// На дисплеї буде відображений символ Q   */
#include <stdio.h>
main()
{
     int  ch = 'Q';
     putchar (ch);
}
```

Зауваження. Слід зазначити, що мова C не має вбудованих операторів введення-виведення, всі ці можливості забезпечує досить багатий комплект стандартних функцій.

І далі обговоримо можливості так званого форматованого введення та виведення, тобто зчитування і виведення інформації у певній заздалегідь визначеній формі.

1. Форматоване стандартне виведення в C.

Форматоване стандартне виведення в C здійснює функція:

```
int printf (char* format, arg1, arg2,..);
```

У неї довільна кількість аргументів **arg1,arg2,..**, повертає вона кількість успішно виведених аргументів. Її призначення – перетворення своїх аргументів у відповідності до змісту стрінгу **format** і виведення їх у стандартний вихідний файл, тобто на екран. Стрінг **format** може містити два види символів: звичайні літери, які безпосередньо копіюються в стандартний вивід, і ***специфікації перетворень***, кожна з яких починається з символа **%** . Кількість аргументів функції **printf** має точно відповідати кількості специфікацій формату. Кожна специфікація завершується спеціальною ***літерою-специфікатором***. Вони наведені в таблиці.

Таблиця літер-специфікаторів

Символ	Тип аргументу	Вид друку
d, i	**int**	десяткове ціле
o	**int**	вісімкове ціле без знаку
x, X	**int**	16-кове ціле без знаку
u	**int**	десяткове ціле без знаку
c	**int**	символ
s	**char***	друкуються символи до **'\0'** або в кількості, визначеною точністю
f	**double**	**[-]m.dddddd** (за замовчуванням точність – 6 знаків)
e, E	**double**	**[-]m.ddddddE±xx**
g, G	**double**	використовує формат **%e** або **%f**
p	**void***	вказівник
%		аргумент не перетворюється, друкується **%**

Отже, специфікація формату має вид:

%[маркер][ширина поля виводу][.точність]літера-специфікатор

Квадратними дужками позначені об’єкти, які можуть бути опущені (разом з дужками). Обов’язковими є лише знак **%** та **літера-специфікатор**. В ролі **маркера** може виступити один із символів:

знак **–** , який вказує, що аргумент при виводі “притискається” до лівого краю поля виводу (за замовчуванням – до правого);

знак **+** , який вказує на необхідність обов’язкового виводу знаку числа (за замовчуванням додатні числа виводяться без знаку **+**);

Під **точністю** розуміється число, що задає точність дійсного числа або мінімальну кількість цифр для цілого числа, або кількість символів стрінга, що виводиться;

Ширина поля виводу визначає кількість позицій на екрані, виділених під виведення відповідного аргументу.

Приклади.

1.
```
// Відпрацьовуємо формати виводу цілої змінної
#include <stdio.h>
main()
{
     int  i = -1;
     printf ("\n i = %d", i); // Поле виводу має вид:    i = -1
     printf ("\n i = %u", i); // Поле виводу має вид:    i = 65535
```

```
     printf ("\n i = %x", i); // Поле виводу має вид:    i = FFFF
     printf ("\n i = %o", i); // Поле виводу має вид:    i = 177777
}
```

Зауваження. Зверніть увагу: одне й те саме значення -1 в залежності від формату має різний вигляд на екрані! Значення константи -1 в різних форматах дані в коментарях для компілятора, в якому **sizeof (int) = 2**. Якщо ж **sizeof (int) = 4**, деякі значення будуть відрізнятись (подумайте: чому, які саме та як саме відрізнятимуться?).

2.
```
// Відпрацьовуємо формат виводу знаку %
#include <stdio.h>
main()
{
     printf ("\n (%%)"); // Поле виводу має вид:    (%)
}
```

Зауваження. Зверніть увагу: для виведення символу **%** його необхідно подвоїти, адже саме таким символом починається специфікація формату, отже ця специфікація має бути «закрита».

3.
```
// Відпрацьовуємо формат виводу дійсної змінної
#include <stdio.h>
main()
{
     float     f = 123.45678;
     printf ("\nf = %8.3f", f); // Поле виводу має вид: f =  123.457
     printf ("\nf = %+8.3f", f); // Поле виводу має вид: f = +123.457
     printf ("\nf = %-8.3f", f);// Поле виводу має вид: f = 123.457
     printf ("\nf = %+12.2f", f);//Поле виводу має вид: f =     +123.46
     printf ("\n f = %7.5fv, f);// Поле виводу має вид: f = 123.45678
}
```

Зауваження. Зверніть увагу на те, як спрацьовують маркери у форматі. Крім того, останній формат є помилковим, адже 7 символів, зазначені як **ширина полю виводу**, очевидно недостатньо, щоб вивести літерал **123.45678**, в якому є 3 символи цілої частини, 5 символів дробової частини (як того вимагає **точність**) та один символ крапка для розділення цілої та дробової частин. Такі невірні формати ігноруються і об'єкт виводиться у повноцінному вигляді.

4.
```
// Відпрацьовуємо формати виводу символьної змінної
#include <stdio.h>
main()
{
     char ch = 'A';
     printf ("\nch = %5c", ch); // Поле виводу має вид:    ch = ____A
     printf ("\nch = %-5c", ch);// Поле виводу має вид:    ch = A____
}
```

Зауваження. Зверніть увагу на різницю у виведенні одного й того самого символу – вона зумовлена **маркером**, адже за умовчанням при виведенні об'єкт «притискається» до правого краю **поля виводу**, тому після знаку рівності перед символом A є 4 вільні позиції у першому виведенні, а у другому – ці вільні позиції залишаться з правого боку від символу A. Вони в коментарях символічно позначені знаками підкреслювання.

5.
```
// Відпрацьовуємо формати виводу стрінгу
#include <stdio.h>
main()
```

```
{
      char*     s = "Holiday";
      printf (s);                 // Поле виводу має вид:   Holiday
      printf ("\n %s", s);        // Поле виводу має вид:   Holiday
      printf ("\n %10s", s);      // Поле виводу має вид:   ___Holiday
      printf ("\n %-10s", s);     // Поле виводу має вид:   Holiday___
      printf ("\n %10.5s", s);    // Поле виводу має вид:   ______Holid
      printf ("\n %-10.5s", s);   // Поле виводу має вид:   Holid______
      printf ("\n %5s", s);       // Поле виводу має вид:   Holiday
}
```

Зауваження. Зверніть увагу, як точність впливає на кількість символів, що виводяться (вільні позиції у полі виводу позначені знаками підкреслювання). Крім того у першому можливі проблеми, якщо стрінг **s** містить знаки **%** або подвійні лапки ", тощо. Отже, самим «безпечним» є другий варіант формату.

2. Форматоване стандартне введення в С.

Форматоване стандартне введення в С здійснює функція:

```
int scanf (char* format, arg1,arg2,..);
```

Вона вибирає символи із стандартного вхідного потоку, інтерпретує їх згідно специфікацій стрінгу **format** і розсилає результати в свої аргументи **arg1,arg2,..**, ***кожен з яких має бути вказівником***. Повертає кількість успішно введених аргументів. Стрінг **format** містить два види об'єктів: звичайні літери, які, як очікується, ***мають з'явитись в стандартному вводі***, і специфікації перетворень, кожна з яких починається з символа **%** . Кількість аргументів функції scanf має відповідати кількості специфікацій формату. Кожна специфікація завершується спеціальною літерою-специфікатором. Вони наведені в наступній таблиці.

Таблиця літер-специфікаторів

Символ	Тип аргументу	Вид вхідних даних
d	**int***	десяткове ціле
i	**int***	будь-яке ціле
o	**int***	вісімкове ціле без знаку
x	**int***	16-кове ціле без знаку
u	**int***	десяткове ціле без знаку
c	**char***	символ
s	**char***	стрінг
f, e, g	**double***	дійсне число
%		знак **%**, ніяке присвоєння не виконується

Отже, специфікація формату має вид:

%[маркер][ширина поля вводу]літера-специфікатор

Квадратними дужками позначені об'єкти, які можуть бути опущені (разом із дужками). Обов'язковими є лише знак **%** та **літера-специфікатор**. В ролі **маркера** може виступити тільки символ зірочка ***** для відміни присвоєння аргументові.

Полем вводу називається послідовність символів стандартного вводу без пробільних літер. Воно продовжується від пробілу до пробілу або обмежене **шириною поля**, якщо остання задана.

Приклади.

1.
```
// Розбираємо саму поширену помилку
#include <stdio.h>
main()
{
      int  i;
      scanf ("%d",i);      // Помилка – аргумент має бути вказівником !
      scanf ("%d",&i);     // Так вірно !
}
```

2.
```
// Вводимо символи
#include <stdio.h>
main()
{
      char ch;
      char Name [10];
      scanf ("%c", &ch);
      scanf ("%1s", &ch);       // такий формат також працює
      scanf ("%s", Name);//Вірно - ідентифікатор масиву є вказівником!
}
```

Зауваження. Зверніть увагу: при введенні символів у масив знак взяття адреси перед ідентифікатором масиву не є обов'язковим, адже ідентифікатор масиву – це константний вказівник на його початок.

3.
```
// Вводимо цілу змінну
#include <stdio.h>
main()
{
      int  i;
      scanf ("%d", &i);    // Введемо 123
      printf ("%i", i);    // Одержимо 123
      scanf ("%i", &i);    // Введемо -456
      printf ("%i", i);    // Одержимо -456
      scanf ("%o", &i);    // Введемо 10
      printf ("%i", i);    // Одержимо 8
      scanf ("%x", &i);    // Введемо 10
      printf ("%i", i);    // Одержимо 16
      scanf ("%u", &i);    // Введемо -1
      printf ("%x", i);    // Одержимо ffffffff
}
```

4.
```
// Використовуємо маркер *
#include <stdio.h>
main()
{
      int  ch;
      // Припустимо, що буде введено 12345
      scanf ("%*2d %1i %*2d", &ch);
      printf ("%i", ch); // Виведеться:  3
}
```

Зауваження. Що відбуватиметься під час виконання? Із прочитаних символів 12345 буде вибрано перші 2 (оскільки вказана **ширина поля вводу = 2**) і поскільки наявний маркер *****, то ці символи будуть проігноровані. Далі буде прочитаний ще один символ – трійка і присвоєній цілій змінній **ch**. Останні 2 символи також будуть вибрані із буфера клавіатури і також проігноровані. Тому на екран буде виведене значення змінної **ch** рівне 3.

3. Потокові введення та виведення в С++.

Мова С++ має додаткові можливості потокових введення та виведення, зручність яких полягає у безформатних можливостях. Проте і засоби форматування також наявні. Потік можна собі уявляти як таку віртуальну трубу, по якій рухається інформація від клавіатури та на екран. Для забезпечення цих можливостей треба підключити директивою препроцесора **include** файл **iostream** . В ньому, зокрема, визначені чотири стандартних потоки:

- **cin** відповідає стандартному вводу (тобто для вводу об'єктів стандартних типів), зв'язаний з клавіатурою;
- **cout** відповідає стандартному виводу (тобто для виводу об'єктів стандартних типів), зв'язаний з дисплеєм;
- **cerr** відповідає стандартному виводу для повідомлень про помилки, зв'язаний з дисплеєм;
- **cprn** відповідає стандартному виводу (тобто для виводу об'єктів стандартних типів), зв'язаний з принтером;

Вивід здійснюється за допомогою перевантаженого оператора **<<** вставки в потік , ввід – з допомогою перевантаженого оператора **>>** виборки з потоку.

Приклади.

1.
```
// Найпростіші операції
#include <iostream>
using namespace std;
main ()
{
   int    n;
   cout << "\n Input n: ";     // Виведеться текст: Input n:
   cin  >> n;                  // Набране на клавіатурі ціле число
                               // пересилається в змінну n
   cout << "\n n = " << n << "\n";
}
```

Призначення операторів **<<** та **>>** легше запам'ятати, якщо вважати, що кожний з них задає напрямок руху інформації: інструкція **cout << n;** переміщає значення змінної **n** в стандартний потік виводу, тобто на екран, а інструкція **cin >> n;** – навпаки із стандартного потоку вводу, тобто клавіатури, спрямовує набране число в змінну **n** .

Операції **<<** та **>>** можна зчеплювати, як, наприклад, в останній інструкції – у вихідний потік будуть послідовно відправлені: текст **" n = "** (переведений на новий рядок), значення змінної **n** та символ переведення курсора на новий рядок.

Введення та виведення з допомогою потоків в попередніх прикладах здійснені безформатно.

Важливо сказати, що перевантажені операції **<<** та **>>** мають не самий низький пріоритет в таблиці пріоритетів, адже в первісному сенсі це операції побітового зсуву. Тому не дивуйтесь побаченому на екрані в наступному прикладі:

Приклади.

2.
```
// Проявляється пріоритет операцій – не економте на дужках!
#include <iostream>
using namespace std;
main ()
{
   int    n1 = 10, n2 = 20;
   cout << "\n Max of 2 numbers "<< n1 <<" and "<< n2 <<" is: ";
   cout << (n1 > n2) ? n1 : n2; << endl; /* маніпулятор endl
     переводить курсор на новий рядок та звільняє буфер вводу */
}
```

На екрані побачимо:

```
Max of 2 numbers 10 and 20 is: 0
```

Справа якраз полягає в тому, що пріоритет умовного виразу нижчий за пріоритет операції **<<** . Тому інструкція

```
cout << (n1 > n2) ? n1 : n2;
```

виконується як

```
(cout << (n1 > n2)) ? n1 : n2;
```

і оскільки результатом перевірки **n1 > n2** є ХИБНІСТЬ, тобто **0**, цей нуль і друкується на екрані. Для правильного виконання цієї інструкції треба використати дужки:

```
cout << ((n1 > n2) ? n1 : n2);
```

Однією із можливостей форматування потоку виведення є використання маніпуляторів – спеціальних команд, які впливають на стан вихідного потоку **cout** . Для їх використання необхідне рідключення заголовочного файлу **iomanip**. Один з маніпуляторів **endl** ми вже використали для переводу курсору на новий рядок. Ось значення ще деяких маніпуляторів: маніпулятори **hex, oct, dec** змінюють формат цілий типу відповідно на 16-ковий, 8-ковий, 10-ковий; **showbase, noshowbase** встановлюють та відміняють виведення перед числом початкових символів **0** або **0x** для позначення основи системи числення. Маніпулятори **uppercase, nouppercase** визначають вигляд 16-кових цифр (відповідно використовуються великі чи маленькі літери для 16-кових цифр). При введенні та виведенні дійсних значень за замовчуванням діє маніпулятор **fixed** (формат з фіксованою десятковою крапкою). Маніпулятор **scientific** змінює цей формат на формат з рухомою десятковою крапкою. Маніпулятор **setprecision(n)** визначає точність **n** знаків (за замовчуванням точність 6). Маніпулятор **setw(n)** встановлює ширину поля виводу, рівною **n** позицій, **left** – вирівнювання виводу по лівій межі, **right** (умовчання) – по правій, **boolalpha** – забезпечує вивід логічних величин в текстовому вигляді (**true, false**) замість вигляду 1 та 0.

Приклади.

3.
```
// Використовуємо маніпулятори
#include <iostream>
#include <iomanip>
using namespace std;
```

```
main ()
{
   int    n = 10;
   double x = 123.45678;
   char c = 'A';
   cout << n << hex << " " << n << oct << " " << n << endl;   // 1
   cout << showbase << n << hex << " " << n << oct << " " << n <<
dec << endl;                                                  // 2
   cout << x << setprecision(2) << fixed << " " << x << endl; // 3
   cout << scientific << setprecision(4) << " " << x << endl; // 4
   cout << c << " " << left << setw(4) << c << "!" << right << setw(4)
<< c << "!" << endl;                                          // 5
}
```

Зауваження. В рядку, поміченому коментарем `// 1`, буде виведено **`10 a 12`** (одне й те саме значення **`10`** буде виведено в числових системах з основами відповідно 10, 16 та 8).
В рядку, поміченому коментарем `// 2`, буде виведено **`10 0xa 012`** (додається префікс для позначення основи системи числення).
В рядку, поміченому коментарем `// 3`, буде виведено **`123.457 123.46`** (одне й те саме значення **`123.45678`** буде виведено в з усіма 6 значущими розрядами та з точністю, встановленою маніпулятором для дійсних чисел у форматі з фіксованою десятковою крапкою).
В рядку, поміченому коментарем `// 4`, буде виведено **`1.2346e+002`** (у форматі з рухомою десятковою крапкою та з вказаною точністю мантиси).
В рядку, поміченому коментарем `// 5`, буде виведено **`A A  !  A!`** (один і той самий символ **`A`** спочатку виводиться безформатно, а потім із вказанням ширини поля виводу = 4 та з притисканням до лівого та правого його країв – символ **`!`** (знак оклику) вказує на кінець поля виводу).

Контрольні питання.

1. Яку мінімальну кількість аргументів може мати функція **`printf`**?
2. Яку мінімальну кількість аргументів може мати функція **`scanf`**?
3. Якими мають бути аргументи функції **`scanf`**?
4. Назвіть потоки стандартних вводу та виводу в C++.
5. Що таке маніпулятори? Навіщо вони використовуються?

Завдання для самоперевірки.

1. Визначити в програмі масив із 10 елементів та ввести їх значення поелементно з клавіатури, вказуючи на екрані індекс кожного елементу, що вводиться (за допомогою засобів мови C).
2. Визначити в програмі масив із 10 елементів та ввести їх значення поелементно з клавіатури, вказуючи на екрані індекс кожного елементу, що вводиться (за допомогою засобів мови C++).

Лекція 5. Функції в C та C++. Класи пам'яті.

Одним із основних понять структурного програмування є модуль – логічно-незалежна частина програми, яка виконує деяку окрему задачу, приховуючи деталі своєї реалізації від інших частин програми. Для підтримки принципу модульності в C/C++ реалізовані ***функції***

(інколи ***метод***) – основні логічно-завершені частини коду. Програма мовою С являє собою сукупність окремих логічно зв'язаних невеликих функцій, очолюваних функцією із зарезервованим ідентифікатором **main**. Саме їй передається управління в першу чергу. В програмі мовою С++ додаються можливості використання класів, в яких функції відіграють роль інтерфейсу. Функції в С/С++ є незалежними, тобто забороняється визначати одну функцію всередині іншої.

1. Визначення функції. Формальні та фактичні параметри.

Визначення функції включає наступні елементи (згідно затвердженому стандарту мови С) :

- вказівку про можливість використання функції поза даним файлом (за допомогою одного з службових слів **static** або **extern**; останнє вживається за замовчуванням);
- тип результату, який повертає функція; відсутність результату позначається службовим словом **void**;
- ідентифікатор (ім'я) функції;
- кількість та типи ***формальних параметрів*** в круглих дужках; в разі їх відсутності вживається службове слово **void**;
- тіло функції, яке взяте в фігурні дужки; складається з визначення внутрішніх (локальних) змінних функції та інструкцій, що мають виконуватись під час роботи функції. Якщо функція повертає результат, то серед цих інструкції обов'язково має бути інструкція виду: **return <вираз>;** причому тип виразу повинен збігатись з типом результату функції.

Таким чином, визначення функції повністю описує інтерфейс та всі дії, що виконуються функцією.

Для того, щоб звернутись до функції (викликати її), досить вказати ідентифікатор даної функції та в круглих дужках перелічити її ***фактичні параметри*** (або ***аргументи***). Кількість та типи фактичних параметрів мають точно відповідати кількості та типам формальних параметрів.

Приклади

1.
```
/* Визначення та використання функції обчислення максимального
елементу в масиві */
#include <iostream>
#include <iomanip>
#include <cstdlib>        // Тут декларації методів srand(), rand ()
#include <ctime>          // Тут декларація методу clock()
using namespace std;
float max ( float* data, int n )
// формальні параметри функції:
// data – вказівник на масив,
// n – кількість елементів
{
   float m;
   int   i;          // m та i – локальні параметри функції max
   m = data [0];     // визначаємо максимумом 0-ий елемент
      for (i = 1; i < n; i++)
```

```
        if (data [i] > m) m = data [i]; // переприсвоюємо максимум,
                        // якщо черговий елемент перевищив його значення
    return m;           // функція max повертає результат
}
main ()
{
    float *real_data, m;
    int n;      // real_data, m та n – локальні параметри функції main
    do
    {
      cout << "Input numbers of elements: n = ";
      cin >> n;
    }
    while (n <= 0); // кількість елементів має бути додатна
    real_data = new float [n];
    srand (clock()); // встановлюємо базу для датчика випадкових
                     // чисел в залежності від часу роботи процесора
    for (int   i = 0; i < n; i++)
    {
                     // елементи масиву - випадкові числа з [0..10]
      real_data [i] = (float) rand ()/RAND_MAX*10;
                     // виводимо елементи масиву з точністю 2 знаки
      cout << setprecision (2) << fixed << real_data [i] << "\t";
    }
    cout << endl;
    // У змінній m буде збережений результат функції
    m = max (real_data, n);     // виклик (invocation) функції,
                                // тут real_data та n –
                                // фактичні параметри
      // Виводимо результат функції:
    cout << "Maximum = " << m << endl;
    return 0;
}
```

Детально розглянемо на цьому прикладі вищезгадані елементи визначення функції.

1. Оскільки при визначенні функції немає вказівок про можливість її використання в інших файлах, які будуть спільно компілюватись з даним, то клас пам'яті (про класи пам'яті дивись нижче) даної функції буде визначений за замовчуванням як **extern**.

2. Тип результату даної функції є **float**. Зверніть увагу на інструкцію **return m;**, вона забезпечує повернення обчисленого функцією значення змінної **m** у точку виклику.

3. Дана функція має ідентифікатор **max**, який відбиває власне її зміст – обчислення максимального елементу в масиві.

4. Функція **max** має 2 формальних параметри: вказівник на масив з дійсними елементами **data** та цілу змінну **n**, рівну кількості елементів масиву. Зауважимо, що можна було описати формальний параметр **data**, як **float data []**.

Параметри в заголовку функції називаються формальними, тому що вони задають формальну схему обробки. Параметри ж, з якими функція викликається, називаються фактичними. Ідентифікатори формальних та фактичних параметрів можуть бути різними, але строга взаємна однозначність між ними – обов'язкова.

5. Тіло функції міститься в фігурних дужках. Перші дві інструкції – це визначення внутрішніх (локальних) змінних функції: дійсна **m** та ціла **i**.

Для того, щоб у функції **main** звернутись до функції **max**, необхідно просто вказати її ідентифікатор з фактичними параметрами. Отже, на момент звернення до функції компілятору має бути відомим все про ім'я **max**. Тобто визначення функції **max** логічно передує визначенню функції **main**. Далі буде з'ясовано, які є інші можливості у структурі коду програми.

2. Особливості передачі параметрів функцій С(С++).

Слід звернути увагу на одну істотну особливість функцій мов С/С++: всі параметри функцій передаються ***за значенням***. Нагадаємо, що технологія передачі параметрів за значенням полягає в тому, що область пам'яті для параметрів функції виділяється в стеку (як для звичайних локальних параметрів функції) та ініціалізується в момент виклику функції значеннями фактичних параметрів. Отже, при виклику функції **max** в стеку буде створена локальна змінна з іменем **n** . Цій змінній буде присвоєне значення одноіменної змінної із функції **main**. Це означає, що функція **max** використовує лише значення змінної **n** із функції **main**, не маючи доступу до самої цієї змінної.

Щоб підсилити розуміння цього факту, розглянемо ще один приклад.

Приклади

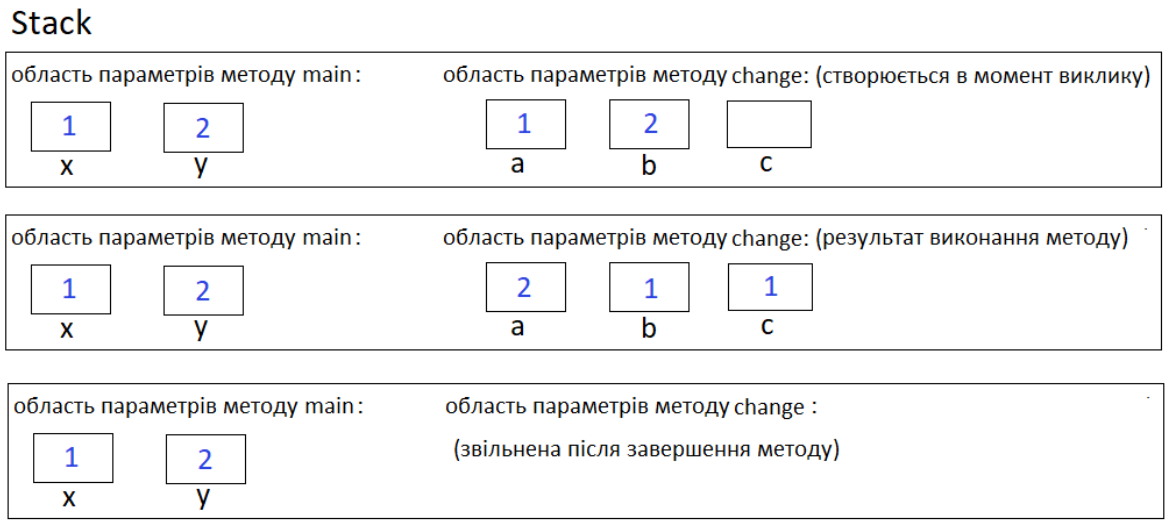
2. А.
```
// метод міняє місцями свої параметри?
#include <iostream>
using namespace std;
void change ( int a, int b )
{
   int   c;
   c = a;
   a = b;
   b = c;
   cout << "\n In change: " << a << " " << b << endl;
}
main ()
{
int   x = 1, y = 2;
   cout << "\n main (before change): " << x << " " << y << endl;
   change (x, y);
   cout << "\n main (after change): " << x << " " << y << endl;
return 0;
}
```

Результатом виконання програми будуть такі повідомлення на екрані:

```
main (before change): 1 2
In change: 2 1
main (after change): 1 2
```

Отже, хоча функція **change** чесно виконала свою задачу – переставила значення змінних, які одержала при виклику, реальні значення змінних в функції main залишились незмінними. А справа полягає саме в передачі параметрів ***за значенням***. Функція дійсно переставила свої

локальні змінні **a** та **b**, які створились під час її виклику в стеку, і не змогла дістатись до фактичних змінних **x** та **y**, які визначені в **main** :



Щоб надати деякій функції можливість оперувати самими змінними, а не їх копіями, треба передавати їй ***значення вказівників на ці змінні***. Внесемо в приклад 2 деякі зміни.

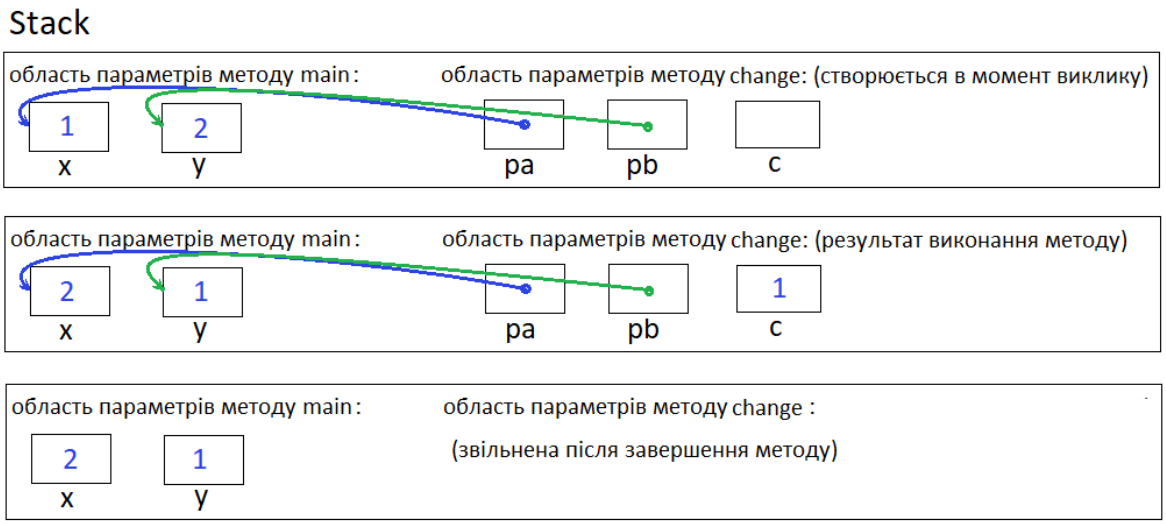
Приклади

2. Б.
```
// метод міняє місцями свої параметри?
#include <iostream>
using namespace std;
void change ( int *pa, int *pb )
{
   int   c;
   c   = *pa;
   *pa = *pb;
   *pb = c;
   printf ("\n In change: %i %i", *pa, *pb);
}
main ()
{
int   x = 1, y = 2;
   cout << "\n main (before change): " << x << " " << y << endl;
   change (&x, &y);
   cout << "\n main (after change): " << x << " " << y << endl;
return 0;
}
```

Цього разу після виконання програми на екрані побачимо такі повідомлення:

```
main (before change): 1 2
In change: 2 1
main (after change): 2 1
```

Тобто тепер, передаючи у функцію **change** в якості фактичних параметрів адреси змінних **x** та **y**, функції надається доступ до них, а отже, можливість їх змінити, що власне і було заплановано:



Повернемось до розгляду формальних параметрів функції **max** з прикладу 1. В списку формальних параметрів окрім кількості елеметів масиву **n** присутній також вказівник на масив **data**. Це означає, що по-перше, копії самого масиву в стеку не створюється, проте створиться копія вказівника на масив, а отже, по-друге, функція має доступ до значень елементів масиву і теоретично може їх змінити. Проте, вона не може замінити цей масив на інший (тобто змінити значення вказівника, адже сам вказівник передається за значенням!). До речі, у функцію можна передати фактичний масив, починаючи з будь-якого елементу, наприклад,

```
m = max (real_data + 2, n - 2);
```

В результаті такого виклику значенням змінної **m** буде максимум серед елементів масиву, починаючи з другого.

3. Області видимості та класи пам'яті.

Дамо означення поняття області видимості об'єкта:

областю видимості (областю дії) імені (об'єкту) називається та частина програми, де це ім'я може використовуватись.

Всі об'єкти (під ними поки що розуміємо змінні та функції) діляться на два види: ***зовнішні*** або ***глобальні*** та ***внутрішні*** або ***локальні***. Зовнішні змінні, як випливає з назви, визначаються зовні будь-якої функції, внутрішні ж визначені всередині деякої функції або навіть блоку. З їх областю видимості все зрозуміло: область видимості локальних об'єктів продовжується від точки визначення цього об'єкту до кінця блоку, тобто функції або складеної інструкції всередині функції. Зрозуміло також, що внутрішніми (локальними) об'єктами можуть бути тільки змінні, бо ***функції завжди є зовнішніми об'єктами***.

Оскільки кожна функція має свою локальну область видимості і кожний блок всередині функції являє собою окрему локальну область видимості, то ці області можуть бути вкладеними.

Визначення функції в глобальній області видимості дає ***глобальну функцію***, а визначення змінної в цій області – ***глобальну змінну***. Глобальний об'єкт існує протягом роботи всієї програми. ***Час життя*** глобального об'єкту починається з моменту запуску програми і закінчується з її завершенням. Час життя об'єкту визначає його ***клас пам'яті***. Класи пам'яті визначені в наступній таблиці.

Зовнішні (глобальні) об'єкти – функції та змінні	Внутрішні (локальні) об'єкти – змінні
extern (за замовчуванням)	**auto** (за замовчуванням)
static	**register**

Будемо розрізняти ***визначення*** функції (обговорювалось вище) від її ***декларації*** (інколи кажуть – ***прототипу*** або ***оголошення***).
Декларація встановлює лише ім'я функції, а також тип результату і типи параметрів.

Необхідність використання декларації функції виникає, наприклад, коли функція визначена в деякому іншому файлі, а в даному файлі треба помістити її ім'я в область видимості файлу або функція визначається в даному ж файлі пізніше, але область видимості потрібно надати їй просто зараз. Визначення будь-якої функції має бути єдиним на всі файли програми, в той час як декларацій в разі потреби може бути декілька – це дія так званого «правила одного визначення».

Службове слово **extern** при визначенні функції або глобальної змінної (зазвичай пропущене за замовчуванням) означає глобальну область видимості для всіх файлів програми. Службове слово **static** при визначенні функції або глобальної змінної означає, що областю видимості даної функції або змінної є лише даний файл. Якщо локальна змінна оголошується з класом **static**, то при всіх викликах функції для збереження її значення буде використаний єдиний статично розміщений в пам'яті об'єкт. Така змінна ініціалізується лише раз при першому виконанні функції. Перевага локальної статичної змінної в тому, що вона дозволяє функції «пам'ятати про минуле» не утворюючи глобальної змінної, яку могли б використовувати (і, можливо, зіпсувати її) інші функції.

Явне визначення класу **auto** для локальних змінних практично не використовується. Службове слово **register** застосовується тільки до локальних змінних або до аргументів функції і означає прохання до компілятора (яке виконується лише про вільних ресурсах) розмістити значення даної змінної безпосередньо на машинних регістрах. Вважається, що така змінна буде інтенсивно використовуватись, тому потрібно прискорити час звертання до неї. Якщо розміщення на регістрах неможливе, змінна просто залишається в пам'яті.

Приклади

3.
```
// Вивчаємо статуси змінних та функцій - класи пам'яті
#include <iostream>
using namespace std;
int global = 111;    // декларація, створення та ініціалізація
                     // глобальної змінної
extern float Glob_A [];  // декларація (не створення!)
                         // глобального масиву
void fun ();         // декларація (не визначення!) функції fun()
int main ()
{
     cout << "1) global = " << global << endl;
     fun ();
     cout << "2) global = " << global << endl;
     // Оскільки масив вже задекларований, його можна використовувати
     Glob_A [0] = 1.5;
     cout << Glob_A [0] << " " << Glob_A [1] << endl;
     {
          // блок – локальна область видимості
```

```
            int global = 555; // декларація, створення та ініціалізація
                  // локальної змінної global, яка перекриває
                  // одноіменну глобальну змінну в межах даного блоку
            cout << "3) global = " << global << endl;
      }     // тут закінчилась область видимості і час існування
            // локальної змінної global
      cout << "4) global = " << global << endl; // тут знову видима
                  // глобальна змінна global
      system ("PAUSE");  // системна затримка
      return 0;
}
void fun ()          // визначення функції fun()
{
      global++;      // Функція змінює значення глобальної змінної
}
float Glob_A [2];  // визначення (створення)
                  // задекларованого вище масива,
                  // його елементи в момент створення будуть рівні 0
```

Зауваження. Результатом запуску цієї програми будуть такі повідомлення:

```
1) global = 111
2) global = 112
1.5 0
3) global = 555
4) global = 112
```

Зверніть увагу: у функції **fun ()** глобальна змінна **global** доступна і її значення змінюється, що може бути небажаним для інших частин програми. Не використовуйте без крайньої і свідомої потреби глобальні змінні для передачі інформації у функції. Зауважте також, що глобальні змінні автоматично ініціалізуються нульовими значеннями. Не використовуйте однакові ідентифікатори для локальних та глобальних змінних, пам'ятайте про ефект «перекриття».

4. Рекурсія.

Рекурсивною називається функція, яка безпосередньо або непрямим чином (тобто через деяку іншу функцію) викликає саму себе.

Розглянемо приклад рекурентної функції (по мотивам прикладу з [2])

Приклади

4.

```
      // Функція itostr перетворює цілу змінну n
      // в послідовність символів, що зображують її
#include <iostream>
using namespace std;
int level = 0;       // змінна level визначає рівень глибини рекурсії
void itostr (int num);  // декларація функції, її визначення - далі
int main ()
{
      int n = 1234567;     // Тестове значення змінної n
      itostr (n);          // Виклик рекурсивної функції для змінної n
      putchar ('\n');      // завершуємо переводом курсора
```

```
      return 0;
}
//////////////////////////////////
void itostr (int num)    // перетворює ціле число в його зображення
{
      if (num < 0)          // якщо число від'ємне, друкуємо мінус і
                            // перетворюємо число в додатне
      {
            putchar ('-');
            num = -num;
      }

      int temp;             // робоча змінна
      if (temp = num/10)
      {
            level++;        // збільшуємо глибину рекурсії
            // друкуємо глибину рекурсії і значення змінної виклику
            cout << "level: "<< level << " " << temp << endl;
            itostr (temp); // ось він - рекурсивний виклик!
      }
      putchar (num%10 + '0'); // після завершення чергового
            // рекурсивного виклику друкуємо чергову цифру числа
      level--; // «випірнаємо» на попередній рівень рекурсії
}
```

Зрозуміло, що в рекурентній функції має бути умова виходу з рекурсії, інакше функція викликатиме саму себе, аж до переповнення стеку. В даному прикладі такою умовою є перевірка на нуль результату ділення **num** на **10** в операторі **if**. Після виходу з кожного рівня рекурсії функція друкує черговий символ числа **num**. Зверніть увагу на передостанній рядок функції **itostr** – функція **putchar** друкує символ, код якого передається їй як аргумент. Тому якщо **num%10** – це чергова цифра числа , то **num%10 + '0'** – це код цієї цифри.

5. Деякі стандартні функції.

Математичні функції (файл **<math.h>** або **<cmath>**)

Далі аргументи **x** та **y** мають тип **double**, **n** – тип **int** ; всі функції повертають значення типу **double** .

sin (x) – синус **x** ;

cos (x) – косинус **x**;

tan (x) – тангенс **x**;

asin (x) – арксинус **x** в діапазоні $\left[-\pi/2, \pi/2\right]$;

acos (x) – арксинус **x** в діапазоні $[0, \pi]$;

atan (x) – арктангенс **x** в діапазоні $\left(-\pi/2, \pi/2\right)$;

atan2 (y, x) – арктангенс y/x в діапазоні $\left(-\pi/2, \pi/2\right)$;

exp (x) – **e** в степені **x** ;

log (x) – логарифм натуральний **x**;

log10(x) – логарифм десятковий **x**;
pow (x,y) – **x** в степені **y** ;
sqrt (x) – корінь квадратний з **x**;
fabs (x) – модуль **x**;
ceil (x) – мінімальне ціле у вигляді **double** , яке більше або рівне **x** ;
floor (x) – мінімальне ціле у вигляді **double** , яке менше або рівне **x**.

Контрольні питання.

1. Визначити поняття області видимості.
2. Визначити поняття класу пам'яті.
3. Яка особливість передачі параметрів в функціях мов C/C++ ?
4. Чим відрізняється визначення функції від її декларації ?
5. Скільки визначень на скільки декларацій функції може бути в програмі?
6. Чим відрізняється визначення змінної від її декларації ?
7. Що таке рекурсія ?

Завдання для самоперевірки.

1. Написати функцію, яка визначає в заданому цілому числі n кількість пар однакових бітів.
2. Написати функцію, яка обчислює факторіал заданого цілого числа n.
3. Написати функцію, яка обчислює дійсні корені квадратного рівняння по його заданих коефіцієнтах a, b, c. В разі відсутності дійсних коренів друкується відповідний результат.
4. Написати функцію, яка посилає користувачеві запит на повторення програми і повертає 1 в разі необхідності повторення програми та 0 в противному випадку.
5. Написати функцію, яка повертає перетворений до верхнього регістру рядок, одержаний як параметр.

Лекція 6. Структури та об'єднання в C та C++.

1. Структури.

Структура в мові C/C++ – це тип даних, який складається з визначеної кількості елементів, що називаються членами структури (інколи – полями структури). Члени структури можуть мати різні типи. Синтаксис визначення структури (у стандарті C):

```
struct tag_name
{
      <тип_1> mem_1;
      <тип_2> mem_2;
            ...
      <тип_n> mem_n;
};
```

Тут **mem_1, mem_2 , … , mem_n** – члени (поля) структури, визначеної з тегом (тобто з іменем, тег – дослівно означає наклейка) **tag_name**. Тепер змінну, яка відноситься до структури такого типу, або, як ще кажуть ***екземпляр*** цієї структури, можна визначити таким чином:

```
struct tag_name id_example;          // стиль C
tag_name id_example;                 // стиль C++
```

Зауваження.

1. При визначенні структури тег не є обов'язковим. Але тоді екземпляри структури, визначеної таким чином анонімно – без тегу, повинні визначатись відразу після фігурної дужки, що закриває тіло структури.
2. При визначенні екземпляру структури в мові С вживання службового слова **struct** обов'язкове, а в С++ – може бути пропущене.
3. Звертання до членів структури відбувається через крапкову нотацію.
4. Обмежень на типи членів структури немає, крім одного – членом структури не може бути екземпляр даної структури, проте може бути вказівник на неї.
5. В С++ членами структури можуть бути функції (методи), зокрема – конструктор структури.
6. Вважається хорошим стилем програмування тег структури записувати великими літерами – так тип структури краще відрізняти від звичайних змінних.

Приклади.

1.
```
// Визначаємо полярні координати на площині
struct POLAR
{
     float r;
     float phi;
}
 // Визначаємо точки a, b , c в полярній системі координат
struct POLAR a, b, c;
```

Кожна із змінних **a, b, c** (екземпляри структури **POLAR**) містить по два поля, це полярні координати – радіус та кут. Щоб дістатись до членів цих екземплярів, необхідно вказати ідентифікатор екземпляру структури та через крапку ідентифікатор поля:

```
a.r   = 1;     a.phi = 0;
b.r   = 4;     b.phi = M_PI;
```

2.
```
// Визначаємо структуру – комплексне число
struct COMPLEX
{
     double Re, Im;
};
// Визначаємо комплексні числа – екземпляри структури complex
// екземпляр x створюється з одночасною ініціалізацією полів
COMPLEX x = {1.0, -1.0}, y, *pz; // В C++ – struct опускаємо
```

Тут визначено два комплексних числа, причому члени першого з них ініціалізовані ***в момент визначення*** відповідно значеннями 1.0 та –1.0, та вказівник **pz** на структуру **complex** .

Допустимі операції з структурами.

Із структурами можна виконувати наступні дії:

- доступ до членів структури ;
- копіювання та присвоєння структур (копіюються або присвоюються відповідні поля)
- взяття адреси структури.

Структури також можуть передаватись в функції як параметрів та повертатись як результат функції. Повернемось до прикладу 2 для ілюстрації дій із структурами.

Приклади.

2.
```
// Продовження
struct COMPLEX
{
```

```
        double Re, Im;
    };
    COMPLEX = {1.0, -1.0}, y, *pz;
    y  = x;          // копіювання структури
    pz = &x;         // взяття адреси структури
    (*pz).Re = 0;  /* дужки обов'язкові, адже пріоритет операції
                      доступу до структури (.) вищий за операцію
                      розадресації (*) */
    (*pz).Im = 1.0;
```

Для спрощення запису доступу до членів структури, що адресується вказівником, використовується спеціальна операція, яка позначається **->** (знак **-** та знак **>**). Таким чином, останні два рядки попереднього прикладу можна записати так:

```
    pz->Re = 0;
    pz->Im = 1.0;
```

Зверніть увагу, ліворуч від знаку операції **->** знаходиться вказівник на структуру, праворуч – член цієї структури.

Приклади.

2.
```
// Продовження
#include <iostream>
#include <cmath>
using namespace std;
struct COMPLEX   // структура реалізує комплексне число
{
    double Re, Im;
};
//  функція додає 2 комплексних числа, результат функції - структура
COMPLEX add (COMPLEX c1, COMPLEX c2)
{
    COMPLEX temp = {c1.Re + c2.Re, c1.Im + c2.Im};
    return temp;         // функція повертає структуру
}
void print (COMPLEX c)    // функція комплексне число
{
    cout << c.Re;
    if (c.Im > 0) cout << " + " << c.Im << "*i" << endl;
    else if (c.Im < 0) cout << " - " << fabs(c.Im) << "*i" << endl;
}
int main ()
{
    COMPLEX a = {1, 2};     // створили і проініціалізували структуру
    COMPLEX b = {-1, -2};  // створили і проініціалізували структуру
    COMPLEX *p = new COMPLEX;  // створили динамічну структуру
    p -> Re = -5;
    p -> Im = 2;
    a = add (a, b);       // виклик функції
    print (a);            // друк результату додавання, він рівний 0
    print (*p);
    return 0;
}
```

Результат виконання цього прикладу буде таким:

```
0
-5 + 2*i
```

Вкладеність структур.

Оскільки структури можуть включати члени довільних типів, природно виникають вкладені структури, тобто структури, полями яких є екземпляри іншої структури. Розглянемо ще один приклад.

Приклади

3.
```
// Вкладеність структур
struct DATE       // структура, що описує дату
{
        unsigned day, month, year;
};
struct PERSON     // структура, що містить інформацію про особу
{
        int    num;
        char   name [20];
        char   address [20];
        DATE birthday;   // поле – структура DATE
};
  struct PERSON *pPr; // вказівник на структуру PERSON
  // створюємо динамічну структуру – виділяємо пам'ять
  pPr = (PERSON *) malloc (sizeof (Person PERSON)); // Стиль C
```

або

```
pPr = new PERSON;                                  // Стиль C++
```

Тепер проініціалізувати члени цієї структури можна, наприклад, таким чином:

```
(*pPr).num  = 1;
```

І оскільки присвоєння масивів неприпустиме, використовуємо функцію **strcpy**, яка копіює перший символьний масив у другий:

```
strcpy ((*pPr).name, "Ivanov I.I.");
```

або через операцію **->** :

```
strcpy (pPr->address,  "no address");
```

Щоб дістатись до полів поля **birthday**, яке в свою чергу є структурою, треба записати ще довшу конструкцію:

```
pPr->birthday.day   = 31;
pPr->birthday.month = 12;
pPr->birthday.year  = 2003;
```

Якщо тепер запустити на виконання цей приклад, додавши до визначення структур **DATE** та **PERSON** метод **main**:

```
#include <iostream>
#include <cstring>
int main ()
{
      struct PERSON *pPr; // вказівник на структуру PERSON
      pPr = new PERSON;   // Ñòèëü Ñ++
      (*pPr).num  = 1;
      strcpy ((*pPr).name, "Ivanov I.I.");
      strcpy (pPr->address,  "no address");
```

```
    pPr->birthday.day = 31;
    pPr->birthday.month =12;
    pPr->birthday.year = 2001;
    cout << pPr->num << " " << pPr->name << " " << pPr->address
    << " " <<pPr->birthday.day <<"/"<< pPr->birthday.month
    << "/" << pPr->birthday.year << endl;
    return 0;
}
```

то одержимо такий результат:

```
1 Ivanov I.I. no address 31/12/2001
```

Масиви структур.

Можна визначати масиви, які мають структури своїми елементами. Наприклад, масив

```
struct PERSON group [25];
```

містить 25 елементів типу структури **PERSON**.

Структури із вказівниками на себе.

Однією з поширених форм організації даних є списки та дерева. Списком називається послідовність структур, зв'язаних в один або в обидва боки між собою. Дерево (бінарне) – це граф, вузлами якого є структури, кожна з яких має не більше двох зв'язків.

В будь-якій з вказаних форм організації даних використовуються структури, членами яких є вказівники на саму структуру. Розглянемо для прикладу однозв'язний список (тобто список, між елементами якого встановлені зв'язки в один бік).

Приклади.

4.
```
// Однозв'язний список – кожна структура містить вказівник last на
попередню  структуру
   struct PERSON
     {
     unsigned key;
     char     name [20];
     struct PERSON *last;  // поле last - вказівник на структуру
                           //даного типу
     };
     /* Функція build_list будує однозв'язний список заданої довжини
     n і повертає вказівник на початок списку*/
   struct PERSON * build_list (unsigned n)
   {
   struct PERSON *p, *q;
   int i;
   q = NULL;
     for (i=1; i<=n; i++)
     {
     p = new PERSON;
        if ( p != NULL )
        {
        p->key  = i;
        p->last = q;
        q       = p;
        }
     }
```

```
return q;
}
```

2. Об’єднання в С (С++).

Об’єднання в С(С++) – це тип даних, який в одній і тій самій області пам’яті може містити (в різні моменти часу) об’єкти різних типів. Це певний аналог варіантної частини записів в Паскалі, якщо вам знайома ця мова. Іншими словами, об’єднання можна визначити як структуру з нульовим зміщенням членів структури відносно початку структури.

Синтаксис визначення об’єднання:

```
union tag_name
{
      <тип_1> mem_1;
      <тип_2> mem_2;
            ...
      <тип_n> mem_n;
};
```

Зауваження

1. Всі зауваження, які були зроблені щодо структур, справедливі і для об'єднань.
2. Ініціалізація об’єднання можлива лише значенням для першого поля.
3. Для екземпляру об’єднання буде виділений такий об’єм пам’яті, щоб в ньому можна було розмістити найбільше з його полів.
4. Об'єднання часто використовуються для неявного приведення типів – наприклад, можна записати одним членом об'єднання дійсне значення, а прочитати з іншого – ціле. Проте такі дії можуть привести до неочікуваних побічних ефектів.

Доступ до членів об’єднання здійснюється так само, як і до членів структури. І взагалі, об’єднання може бути членом структури та навпаки.

Приклади.

6.
```
// Вивчаємо об’єднання
#include <iostream>
using namespace std;
union CONVERT
{
     unsigned n;
     char c;
     int i;
     int* p;
};
int main ()
{
    CONVERT conv = {65};
    cout << conv.n << " " << conv.c << " " << conv.i << " "
         << conv.p << endl;
    conv.c = '0';
    cout << conv.n << " " << conv.c << " " << conv.i << " "
         << conv.p << endl;
    conv.i = -1;
    cout << conv.n << " " << conv.c << " " << conv.i << " "
         << conv.p << endl;
    return 0;
```

```
}
```

Зауваження. Початкова ініціалізація значенням **65** стосується поля **n**. При виведенні на екран значень всіх інших полів стає зрозумілим, що символьне поле є символом із кодом **65**, тобто це літера **A**. Поле **i** буде так само мати значення **65**, а от для поля **p**, яке є вказівником, це значення буде проінтерпретоване як адреса, тобто це буде шістнадцяткове число **41**.

Переконайтесь, що результат виконання даного прикладу буде таким:

```
65 A 65 0x41
48 0 48 0x30
4294967295   -1 0xffffffff
```

5. Переліки в C (C++).

Перелік (або зліченний тип) – це унікальний тип, значення якого покриваються множиною іменованих констант, які, власне, і утворюють перелік. Синтаксис визначення переліків схожий на визначення структур та об'єднань:

```
enum tag_enum {<перелік констант>};
```

Ідентифікатори всіх констант в переліках мають бути унікальними. Значення констант можна проініціалізувати. Якщо не всі константи ініціалізовані, то вони визначаються за правилом арифметичної прогресії із різницею прогресії 1, починаючи від останньої визначеної константи. За замовчуванням значенням першої константи в переліку є 0, наступних – відповідно 1, 2 і т.д.

Приклади

6.
```
// Визначаємо перелік місяців року
enum MONTHS {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
     OCT,NOV, DEC };
```

Тут **FEB** – це **2**, **MAR** – це **3** і т.д.

Визначення змінної-переліку цього типу може виглядає так:

```
enum MONTHS m; // m може мати значення однієї з констант переліку
```

Для неї допустимі, наприклад, такі присвоєння:

```
m = DEC;
m = (MONTHS) 10;
```

Приклади

7.
```
// Визначаємо перелік, що задає одиниці в певних бітах
#include <iostream>
using namespace std;
enum MASKS // Це одиниці відповідно в 1-му, 2-му, 3-му та 4-му бітах
{
     bit1 = 0X1, bit2 = 0X2, bit3 = 0X4, bit4 = 0X8
};
int main ()
{
    int mask = bit1 | bit3;    //Одержали значення маски:  0....0101
    int num = 7;
    num = num & mask;          // Виконуємо бітову операцію
    cout << num << endl;       // Тут буде виведено 5
    MASKS m = bit1;
    m = (MASKS) 8;
    cout << m << endl;         // Тут буде виведено 8
```

```
    return 0;
}
```
Слід віддати перевагу саме такому способу задання констант на відміну від визначення їх за допомогою макровизначень **#define**.

Контрольні питання.

1. Визначити, що таке структура, об'єднання, перелік.
2. Які типи є допустимими для членів структур та об'єднань?
3. Які операції можна виконувати з структурами?

Завдання для самопідготовки.

1. Написати функцію, яка додає два заданих комплексних числа та повертає їх суму.
2. Написати функцію, яка множить два заданих комплексних числа та повертає їх добуток.
3. Написати функцію, яка ділить два заданих комплексних числа та повертає їх частку.
4. Написати функцію, яка повертає модуль заданого комплексного числа.
5. (*)Написати функцію, яка повертає вказівник на структуру **PERSON** однозв'язного списку із заданим значенням ключа **key** рівним **k0**.

Лекція 7. Робота з файлами в C та C++.

1. Файли в C.

Основним призначенням переважної більшості програм є обробка інформації. Тому важливими проблемами стають збереження та перенесення інформації на зовнішніх носіях – зовнішніх пристроях пам'яті (ЗПП).

Окремий іменований набір інформаційних записів, утворений на зовнішньому пристрої, називається ***файлом*** (фізичним). Доступ до елементів файлу завжди здійснюється послідовно.

З терміном ***файл*** тісно пов'язаний термін ***потік*** – це послідовність даних, що поступає на вхід (вхідний потік) або на вихід (вихідний потік) певної системи або програми. Саме слово ***потік*** підкреслює саме послідовний характер доступу до окремих елементів цієї інформаційної структури. В нашому викладі ці терміни часто можуть вживатись як синоніми.

Основні операції з інформаційними потоками (файлами) такі:

- зчитування чергового запису з потоку в оперативну пам'ять (ОП);
- занесення із ОП запису в потік;
- оновлення запису в потоку.

Як вже згадувалось в лекції 4, в мові C/C++ визначені три стандартних потоки: **stdin** – вхідний потік, зв'язаний із клавіатурою, **stdout** – вихідний потік, зв'язаний з дисплеєм та **stderr** – вихідний потік помилок, також зв'язаний з дисплеєм. Тут буде розглядатись використання файлів (файлових потоків) відмінних від стандартних.

Отже, рух інформації між зовнішнім пристроєм та оперативною пам'ятью схематично (дуже приблизно) можна уявляти собі таким чином:

ЗПП ←----------→ **ОП**

файл потік програма

Для узгодження швидкостей операцій читання-запису потік рухається через буфер. ***Буфер*** – це робоча область пам'яті, що використовується для тимчасового збереження

інформації, коли швидкість обміну між ЗПП та ОП менша за швидкість операцій читання-запису.

Для керування роботою з файлами програма повинна повідомляти системі певну інформацію про файл, як мінімум, його ідентифікатор, повний шлях до нього, характер роботи з файлом (читання чи запис), місце знаходження поточного запису в файлі, чи були попередні помилки при роботі з файлом, тощо. Процедура початкової фіксації цієї інформації називається ***відкриттям файлу***. Отже, будь-який файл перед виконанням першої операції з ним має бути відкритим з допомогою стандартної функції **fopen** , яка заносить початкову інформацію про файл в стандартну структуру з іменем **FILE** .

Декларація цієї стандартної функції має вид:

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

Перший параметр – це стрінг, який має містити повне ім'я файлу (із вказанням повного шляху, або ж в разі лише ідентифікатору файлу останній має знаходитись у поточній директорії, тобто тій, звідки запускається файл на виконання), другий параметр, також стрінг, визначає режим роботи з файлом, який може бути одним з наступних:

- **"r"** – відкрити існуючий файл для вводу з нього (читання);
- **"w"** – створити новий файл або відкрити (та звільнити від попереднього вмісту) існуючий файл для виводу (запису) в його початок;
- **"a"** – створити новий файл для виводу або відкрити існуючий файл для виводу в кінець файлу;
- **"r+"** – відкрити існуючий файл для оновлення (тобто читання та запису), яке буде здійснюватись з його початку;
- **"w+"** – створити новий файл або відкрити існуючий файл для оновлення, яке буде здійснюватись з його початку;
- **"a+"** – створити новий файл або відкрити існуючий файл та підстроїтись в його кінець для оновлення.

В разі успішного відкриття файлу ця функція повертає так звану файлову змінну – вказівник на структуру **FILE**; в разі виникнення помилки при відкритті файлу ця функція повертає **NULL**. Якщо до символів, які задають режим, дописана літера **t** , то файл використовується як текстовий (цей режим діє за замовчуванням), якщо ж літера **b** – то як двійковий.

Зауваження. Хоча режим оновлення допускає як запис так і читання з файлу, але при переході від однієї операції до іншої слід звертатись до стандартної функції

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

яка виконує дозапис всіх даних, що ще залишились в буферах.

Для закриття файлу (тобто звільнення буферів і файлової змінної та знищення інформації про файл) використовується стандартна процедура з декларацією:

```
int fclose (FILE* fp);
```

Ця процедура повертає **0** в разі успішного завершення та **EOF** (стандартна константа, визначена в **stdio.h**) – в разі помилки. Оскільки кількість одночасно відкритих файлів обмежена можливостями операційної системи, необхідно зайві файли вчасно закривати.

Приклади

1.
```
// Відкриваємо для запису в початок файл my_file.txt на диску A
#include <stdio.h>
void main ()
{
    FILE *fp;
```

```
        fp = fopen ("A:\\my_file.txt", "w"); // Зверніть увагу на  \\
        if (fp != NULL)
        {
              // Тут працюємо з  файлом
              /* ....... */
              // після завершення роботи з файлом – закриваємо його:
              fclose (fp);
        }
}
```

2. Деякі стандартні функції для роботи з файлами.

`int fgetc (FILE*fp);`

Здійснює введення чергового символу із вказаного файлу. Якщо повертає **EOF**, то сталась помилка або досягнуто кінця файлу.

`int fputc (int ch, FILE*fp);`

Виводить символ з кодом **ch** у вказаний файл. Повертає змінну з кодом **ch** або **EOF**, якщо сталась помилка.

`char * fgets (char *s, int n, FILE *fp);`

Ця функція здійснює введення в стрінг **s** (буфер) символів із файлу, який ідентифікується файловою змінною **fp**, допоки не виконається одна з умов:

- виникає символ **'\n'** (початок нового рядку);
- досягнутий кінець файлу;
- прочитано **n-1** символів.

Після цього стрінг доповнюється **'\0'** . В першому випадку перед **'\0'** записується **'\n'**. Якщо читання з файлу завершено успішно, то повертає вказівник на стрінг **s**, в разі помилки повертає **NULL**.

`int fputs (char *s, FILE*fp);`

Ця функція виводить стрінг **s** у вказаний файл. Повертає **0** або **EOF**, якщо сталась помилка.

`int fprintf (FILE *fp , char *format, …);`

Ця функція здійснює форматований вивід своїх аргументів в файл, який ідентифікується файловою змінною **fp**, під керівництвом стрінгу **format**. Відносно цієї функції справедливо все те, що було сказано про функцію стандартного форматованого виводу **printf** (див. раніше).

`int fscanf (FILE *fp , char *format, …);`

Здійснює форматоване, визначене стрінгом **format**, введення з вказаного файлу **fp**. Відносно цієї функції справедливо все те, що було сказано про функцію стандартного форматованого виводу **scanf** (див. раніше).

`int feof (FILE *fp);`

Повертає не **0** (тобто **ІСТИНУ**), якщо при читанні з файлу був досягнутий кінець файлу і **0** (тобто **ХИБНІСТЬ**) у супротивному випадку.

```
int ferror (FILE *fp);
```
Повертає не **0** (тобто **ІСТИНУ**), якщо при виконанні операцій введення або виведення з файлом **fp** виникали помилки і **0** (тобто **ХИБНІСТЬ**) у супротивному випадку.

```
void rewind (FILE *fp);
```
Пересуває вказівник поточної позиції в файлі на початок файлу. При цьому вказівники кінця файлу та помилок зануляються. Після виклику цієї функції файл можна читати повторно спочатку.

Наступні дві функції працюють із бінарними файлами і забезпечують введення та виведення інформації в «закодованому» у двійковий формат вигляд:

```
size_t fread (void* ptr, size_t size,
              size_t num, FILE* filename);
```
Читає із файлу **filename** в масив **ptr** не більше **num** елементів розміру **size**. Повертає кількість успішно прочитаних елементів (вона може бути менше заявленої).

```
size_t fwrite (void* ptr, size_t size,
              size_t num, FILE* filename);
```
Записує у файл **filename** із масиву **ptr** **num** елементів розміру **size.** Повертає кількість успішно записаних елементів.

Приклади

2.
```
//Копіюємо зміст файлу в інший файл
#include <stdio.h>
int main ()
{
   FILE *f1 = fopen ("in.txt", "r");  // створили файлову змінну f1,
// прив'язали до неї структуру з інформацією про існуючий файл in.txt
// - це і називається відкриттям файлу
   FILE *f2 = fopen ("out.txt", "w"); // файлова змінна f2 -
// тут створиться або відкриється файл out.txt для запису в нього
       if (f1 && f2)  // якщо файли успішно відкриті
       {
           int ch;    // робоча змінна для читання та запису
           while ((ch = fgetc (f1)) != EOF) // прочитали символ і
                    // перевірили, чи не досягнутий кінець файлу
           {
               fputc (ch, f2);  // записали символ ch у вихідний файл
               putchar (ch);    // вивели символ ch на екран
           }
           fclose (f1);   fclose (f2);   // закрили файли
           putchar ('\n');
       }
       else         // якщо проблема з відкриттям одного з файлів
           printf ("Error!\n");
   return 0;
}
```

Зауваження. Зверніть увагу – для читання та запису символів використовуємо робочу змінну **ch** типу **int**. Запам'ятайте цей прийом, адже символьний тип є підмножиною цілого, але системна константа **EOF** (зазвичай її значення рівне **-1**) не є символом з кодової таблиці.

Приклади

3.
```
     // Перевіряємо наявність помилок при читанні
     // з файлу my_file на диску C
#include <stdio.h>
int main ()
{
FILE *fp;
int er;
   fp = fopen ("in.txt", "r"); // Файл відкривається для читання
     if (fp != NULL)           // Якщо файл відкритий
     {
          fprintf ( fp, "\n %s", "We are wrong!"); // Помилка –
               //  спроба писати у файл, відкритий для читання
          if (er = ferror (fp))         // чи зафіксована помилка?
               fprintf (stderr, "%i", er);   // виводимо код помилки
          fclose (fp);                       // закриваємо файл
     }
     return 0;
}
```

Приклади

4.
```
/* Читаємо слова з файлу in.txt . Копіюємо їх в новий файл  out.txt,
починаючи кожне слово з великої букви */
#include <stdio.h>
// Ця функція підносить латинський символ на верхній регістр
// (можна використати системну функцію)
int upper (int c)
{
  if (c >= 'a' and c <= 'z')
     return c + 'A' - 'a';
  else
     return c;
}
int main ()
{
FILE *fp_in, *fp_out;     // декларуємо файлові змінні
int c1, c2;               // робочі змінні для читання
   fp_in  = fopen ("in.txt", "r"); // відкриття файлу для читання
   fp_out = fopen ("out.txt", "w");// відкриття файлу для запису
     if (fp_in && fp_out)          // якщо файли успішно відкриті
     {
          c1 = fgetc (fp_in);      // читаємо перший символ
          while (c1 != EOF)        // чи не досягнутий кінець файлу?
          {
               c2 = fgetc (fp_in); // читаємо наступний символ
```

```
            if ((c1 ==' ') && (c2 != ' ')) // c2 – початок слова?
                 c2 = upper (c2);// підносимо на верхній регістр
            fputc (c1, fp_out);  // записуємо перший символ
            c1 = c2;             // другий символ стає першим
        }                        // кінець циклу while
        fclose (fp_in  );        // закрили вхідний файл
        fclose (fp_out);         // закрили вихідний файл
    }
    else                         // якщо файли не були відкриті
       fprintf (stderr, "\n Error!!!");
    return 0;
}
```

Зауваження. Для тестування у файл **in.txt** був записаний текст

```
aa  bbb   cccccc  qwert   S3245  D
```

Після виконання програми у файлі **out.txt** був такий вміст:

```
Aa  Bbb   Cccccc  Qwert   S3245  D
```

Приклади

5.

```
    // Використаємо функції fread та fwrite для збереження структур
    // в двійковому форматі
#include <cstdlib>
#include <cstdio>
#define NUM_TOV 3         // Визначаємо кількість товарів
using namespace std;      // Закріплення простору імен std
typedef long MY;
typedef struct       // Структура для збереження інформації про товар
{
    string nazva_tov;
    int    kod_tov;
} PRODUCT;
// PRODUCT – визначене операцією typedef назва типу структури
int main ()
{
    PRODUCT arr [NUM_TOV];//Масив з 3-х товарів, описаних структурами
    for (int i = 0; i < NUM_TOV; i++)    // Заповнюємо з клавіатури
                                         // масив структур
    {
          cout << "input tovar: ";
          cin >> arr[i].nazva_tov;
          cout << "input kod: ";
          cin >> arr[i].kod_tov;
     }
     FILE *f = fopen ("tovar.bin", "wb");   // відкриття бінарного
                                            // файлу для запису
     fwrite (arr, sizeof(PRODUCT), NUM_TOV, f);   // прямий запис
                                 // масиву структур у бінарний файл
     fclose (f);                 // закрили файл
     PRODUCT arr_out [NUM_TOV];   // створили масив масив для 3
     // товарів описані структурами, які будуть вводитись із файлу
```

```
f = fopen ("tovar.bin", "rb");    // відкриття бінарного
                                  // файлу для читання
fread (arr, sizeof(PRODUCT), NUM_TOV, f);   // пряме читання з
                            // бінарного файлу у масив структур
for (int i = 0; i < NUM_TOV; i++)   // вивід на екран
                                    // введеного масиву структур
{
   cout << arr[i].nazva_tov << " " << arr[i].kod_tov  << endl;
}
return 0;
```

Зауваження. В ході виконання цього прикладу масив структур буде записаний як суцільний блок інформації у двійковий файл, де зберігатиметься у «кодованому» вигляді, а потім так само блоком, буде прочитаний у інший масив структур. Вивід на екран підтверджує, що інформація була «закодована» та «розкодована» успішно.

3. Файли прямого доступу.

Хоча доступ до елементів файлу послідовний (тобто до кожного елементу файлу можна дістатись, просуваючись від початку файлу), деякі стандартні функції дозволяють організувати роботу з файлами в режимі так званого ***прямого доступу***. Розглянемо ці функції.

```
int fgetpos (FILE *fp, fpos_t *pos);
```

Ця функція заносить в змінну **pos** , типом якої є **fpos_t** (це еквівалент типу **long int**), значення поточної позиції в файлі **fp**. Повертає **0**, якщо позиція **pos** успішно визначена.

```
int fsetpos (FILE *fp, fpos_t *pos);
```

Встановлює в файлі **fp** поточну позицію в положення, визначене вказівником **pos**. Останній мав бути раніше визначений функцією **fgetpos**. Повертає **0**, якщо позиція **pos** успішно встановлена.

```
int fseek (FILE *fp, long offset, int from_where);
```

Встановлює в файлі **fp** поточну позицію в положення, що знаходиться на **offset** байтів далі від положення, визначеного аргументом **from_where** в сторону кінця файлу при **offset>0** і в сторону початку файлу при **offset<0.** Аргумент **from_where** може набувати значення однієї з трьох стандартних констант: **SEEK_SET (=0)** – відлік позиції відбувається від початку файлу; **SEEK_CUR (=1)** – відлік позиції від поточної позиції в файлі; **SEEK_END (=3)** – відлік позиції від кінця файлу. Повертає **0**, якщо позиція успішно встановлена.

Приклади

6.
```
// Реалізуємо прямий доступ до файлу work.txt функцією  fseek
#include <stdio.h>
int main ()
{
     FILE *fp_out = fopen ("work.txt", "w");
     if (fp_ out != NULL)
     {
          fprintf ( fp_out, "\n %s","There is an example.");
          fseek (fp_out, -1L, SEEK_CUR); // Повертаємось на один байт
```

```
                                        // назад  від поточної позиції
            fprintf ( fp_out, "\n %1s","!"); // Замінюємо записану
                                        // крапку на знак оклику
            fclose (fp_out);
      }
      return 0;
}
```

Приклади

7.
```
// Реалізуємо прямий доступ до файлу з функціями fgetpos, fsetpos
#include <stdio.h>
#include <cstdlib>
int main ()
{
      fpos_t pos;
      int    ch;
      FILE *fp_out = fopen ("work.txt", "w");
      if (fp_out != NULL)
      {
            fprintf ( fp_out, "\n Hello, ");
            fgetpos (fp_out, &pos); // Запам'ятаємо поточну позицію
            fflush (fp_out);    // "підштовхнули" вивід з буфера у файл
            system ("PAUSE");  // 1 затримка - перевіримо вміст файлу
            fprintf ( fp_out, "Anna!"); // Дописуємо  текст в файл
            fflush (fp_out);    // "підштовхнули" вивід з буфера у файл
            system ("PAUSE");  // 2 затримка - перевіримо вміст файлу
            fsetpos (fp_out, &pos); // Встановлюємо попередню позицію
            fprintf ( fp_out, "Marya!"); // Дописуємо новий текст
            fflush (fp_out);    // "підштовхнули" вивід з буфера у файл
            system ("PAUSE");  // 3 затримка - перевіримо вміст файлу
            fclose (fp_out);    // закрили файл
      }
}
```

Зауваження. Якщо перевіряти вміст файлу під час затримок 1-3, можна переконатись, що встановлюється попередньо зафіксована позиція **pos**, та відбуваються відповідні зміни (міняється текст **"Anna!"** на **"Marya!"**) у файлі. В цьому прикладі важливу роль відіграє функція **fflush**, без неї вивід у файл, скоріше за все відбувся би лише у момент закриття файлу – до цього текстова інформація знаходилась би у буфері виводу.

4. Прості операції з файлами в C++.

Якщо необхідно читати або записувати текстову інформацію у файл засобами мови C++, то необхідно створити екземпляр потоку введення або відповідно виведення і далі працювати з цим потоком подібно до роботи із стандартними потоками **cin** та **cout**. Більш детально роботу з потоками розберемо дещо пізніше – після знайомства з класами, зараз же розглянемо найпростіші операції. Отже, для читання із файлу виконуємо такі дії:

1. Створюємо екземпляр класу **ifstream**, для керування потоком введення:
   ```
   ifstream ifile;
   ```
2. Ставимо цей екземпляр у відповідність конкретному файлу (наприклад, файлу **in.txt** на диску **D**):

```
ifile.open ("D:\\in.txt");
```

(Кроки 1 та 2 можна замінити створенням екземпляру з допомогою конкретного конструктора:

```
ifstream ifile ("D:\\in.txt", ios::in);)
```

3. Працюємо із цим об'єктом так само, як із об'єктом **cin**, використовуючи операції виборки з потоку **>>**.
4. Закриваємо файл **ifile.close ().**

Для запису у файл виконуємо такі дії:

1. Створюємо екземпляр класу **ofstream**, для керування потоком виведення:

```
ofstream ofile;
```

2. Ставимо цей екземпляр у відповідність конкретному файлу (наприклад, файлу **out.txt** на диску **D**):

```
ofile.open ("D:\\out.txt");
```

(Кроки 1 та 2 можна замінити створенням екземпляру з допомогою конкретного конструктора:

```
ofstream ofile ("D:\\in.txt", ios::out);)
```

3. Працюємо із цим об'єктом так само, як із об'єктом **cout**, використовуючи операції вставки у потік **<<**.
4. Закриваємо файл **ofile.close ().**

Приклади

8.

```
// Переписуємо символи із одного файлу в інший з їх підрахунком
#include <iostream>
#include <fstream>  // підключили файл заголовків методів і класів,
                    // пов'язаних із файловими потоками в С++
using namespace std;  // закріплення простору імен std
int main ()
{
    ifstream f1 ("file_1.txt");// відкрили файловий потік для читання
    ofstream f2 ("file_2.txt");// відкрили файловий потік для запису
    char ch;                   // робоча змінна
    int counter = 0;           // лічильник прочитаних символів
    f1 >> ch;                  // прочитали перший символ
    while (!f1.eof ())         // це не кінець файлу?
    {
     f2 << ch;                 // записуємо символ у вихідний потік
     counter++;
     f1 >> ch;                 // прочитали наступний символ
     }
     cout << "\nC++ Number of symbols: " << counter << endl;
     f1.close ();                    // закрили вхідний потік
     f2.close ();                    // закрили вихідний потік
     return 0;
}
```

Контрольні питання.

1. Визначити, що таке файл, потік, буфер.

2. Що означають операції відкриття та закриття файлу?
3. Що таке файлова змінна, **EOF**?
4. Які стандартні функції здійснюють читання та запис в файл?
5. Що таке файли прямого та послідовного доступів?

Завдання для самоперевірки.

1. Написати програму, яка створює на диску файл із заданим іменем та виводить в нього **n** введених користувачем з клавіатури чисел.
2. Написати програму, яка дописує в файл з заданим іменем **n** введених користувачем з клавіатури чисел.
3. Написати програму, яка виводить на екран вміст файлу із заданим іменем.

Лекція 8. Директиви препроцесора. Інструкція визначення типів typedef.

Впорядкуємо зараз свої відомості про команди (директиви) препроцесора. Препроцесор – це спеціальна програма, яка аналізує код вашої програми раніше компілятора, вибираючи та виконуючи рядки, які починаються із символа **#**. Ми вже добре знайомі з директивою **#include**. Вона забезпечує включення так званих заголовочних файлів (**header-files**), які містять необхідні декларації функцій, констант та класів, щоб забезпечити повною інформацією компілятор. Далі розглянемо ще деякі директиви препроцесора.

1. Макровизначення.

Макровизначення (або макропідстановка, або просто макрос) це ідентифікатор, значення якого визначається директивою **#define**. Препроцесор замінить в усьому коді, який знаходиться після макровизначення, всі включення ідентифікатора на відповідне макровизначення. Вигляд директиви:

```
#define <ідентифікатор>  <текст макросу>
```

Приклади

1.
```
// макрос  SIZE визначає розмірність масиву
#define SIZE 10
int main ()
{
   int arr [SIZE]; // SIZE замінено на 10
   // інші інструкції програми
}
```

Зауваження

1. Зверніть увагу – текст макросу просто «підставляється», тому макровизначення виду:
   ```
   #define SIZE = 10
   ```
   або
   ```
   #define SIZE 10;
   ```
   є помилковими. (Поясніть собі – чому?)
2. Допускається визначення макросів з параметрами – звертання до них виглядає як звертання до функцій. Багато стандартних «функцій» бібліотек C/C++ насправді є макровизначеннями.

3. Директива **#undef <ідентифікатор>** зупиняє дію **ідентифікатора**, який був визначений раніше і дозволяє його повторне визначення в інших цілях.

З'ясуємо, як виглядає згадене вище макровизначення з параметрами. Вигляд відповідної директиви такий:

#define <ідентифікатор>(<параметри>) <текст макросу>

Важливо зауважити, що у визначенні макросу між ідентифікатором та круглою дужкою, в якій містяться параметри макросу, неприпустимий пробіл – адже він означитиме завершення ідентифікатору макросу.

Приклади

2.
```
// розглядаємо макровизначення з параметром
#include <iostream>
#define cube(x) x*x*x // макрос з параметрами визначає куб аргументу
using namespace std;
int main ()
{
    int i = 1;
    double x = 1.0;
    cout << cube (i) << endl;       // виклик універсальний для
    cout << cube (x) << endl;       // всіх типів параметрів!
    cout << cube (i + 1) << endl;   // А що буде виведено тут?
    return 0;
}
#undef cube
```

Зауваження. Як ви поясните результат, рівний 4, замість очікуваного 8, в останньому виклику макросу? А справа полягає в тому, що текст макросу буквально підставляється в код, тобто виклик **cube (i + 1)** буде підставлений так: **i + 1*i + 1*i + 1**, що дійсно рівно 4!

Звідси правило: у визначенні макросу параметр завжди має бути взятий у дужки. Саме значення макросу також має бути в дужках, тому що в загальному випадку він може бути частиною виразу з невідомими пріоритетами операцій. Отже, наведемо правильне визначення макросу з цього прикладу, а також ще декілька макросів із параметрами:

```
#define cube(x) ((x)*(x)*(x))
#define min(x, y) (((x) < (y)) ? (x) : (y))
#define upper(x)((char)((('a'<=(x)) && ((x)<='z'))?(x)+'A'-'a':(x)))
```

Спробуйте з'ясувати призначення останніх двох макровизначень.

Наостанок підсумуємо плюси та мінуси використання макросів. Перший великий і безперечний плюс полягає в тому, що макрос є інваріантним до типу своїх параметрів, тому немає необхідності створювати однотипні функції, які розрізняються лише типами параметрів. По-друге, при звертанні до макросу не витрачається час на виклик та створення в стеку параметрів, як це було б при звертанні до функції. Неприємним є надмірна кількість дужок, яка необхідна при визначенні макросу. І проблемним є те, що використання макросу може приводити до помилок, якщо, наприклад, у звертанні до нього використовуються інкремент або декремент. Перевірте, як спрацює макрос **cube** при виклику **cube (i++)** і переконайтесь у наявній проблемі. Проте, в мовах C/C++ макроси є широко вживаними, багато системних «функцій» насправді є макровизначеннями.

2. Директиви компіляції.

Вже згадувалась директива препроцесора, пов'язана з компіляцією програмних файлів. Для підключення так званих *header-файлів* (файлів заголовків), тобто файлів, які містять визначення констант та макросів, декларації функцій, класів, шаблонів, тощо, використовуються директиви

```
#include <ідентифікатор header-файлу>
```

та

```
#include "ідентифікатор header-файлу"
```

У першому випадку відповідний файл має знаходитись у системних директоріях, а в другому – ***у поточній директорії***, яка і містить файл, що компілюється.

Наприклад, ви можете розмістити всі декларації функцій, які використовуються у вашій функції main, в окремому файлі, розташувавши його в тій же директорії, і підключити цей файл директивою препроцесора **#include** в останньому форматі.

Ще одна група директив використовується для так званої умовної компіляції, тобто для керування роботою препроцесора. Це директиви **#if**, **#elif**, **#else** та **#endif**. Наприклад, наступна група директив **#if**

```
#if константний_вираз_1
// тут деякі директиви препроцесора
     #elif константний_вираз_2
// тут деякі директиви препроцесора
     #else
// тут деякі директиви препроцесора
#endif
```

буде працювати таким чином:

якщо цілий **константний_вираз_1** у директиві **#if** має ненульове значення (**ІСТИНА**), то при компіляції включаються всі наступні (**#elif** рядки до **#elif** (якщо ця директива наявна) або до **#endif** (якщо директив **#elif** та **#else** немає). Тобто директива **#elif** діє як гілка звичайного розгалуження **else-if**.

Приклади

3.
```
// Змінюючи значення константи VERSION, керуємо включенням файлів:
#define VERSION 3
#if VERSION == 1
      #define INCLUDE_FILE "file_1.h"
   #elif VERSION == 2
      #define INCLUDE_FILE "file_2.h"
   #else
      #define INCLUDE_FILE "file_3.h"
#endif
#include INCLUDE_FILE
```

Зауваження. В цьому прикладі значення константи **VERSION** наразі встановленим рівним 3. За логікою директив умовної компіляції, макрос **INCLUDE_FILE** матиме значення **"file_3.h"**. Тому остання директива **#include**, в якій фігурує цей макрос, приведе до включення у код програми заголовочного файлу **file_3.h**. Таким чином, змінюючи значення константи **VERSION**, можна підключати різні заголовочні файли.

Ще один група директив дозволяє позбавитись при компіляції повторних включень заголовочних файлів. Це директиви, які ми пізніше використаємо при створенні повноцінного проекту з декількох файлів:

```
#ifndef <ідентифікатор>
#define <ідентифікатор>
// код, який включається у компіляцію
#endif
```

Перша директива перевіряє, чи був визначений **<ідентифікатор>** директивою **#define**. Якщо ні, то наступна директива його визначає і при компіляції буде включений весь текст до директиви **#endif**. Якщо ж **<ідентифікатор>** вже був визначений, то код між директивами **#define** та **#endif** буде проігнорований.

3. Інструкція визначення типів typedef

Інструкція **typedef** дозволяє визначати нове, альтернативне ім'я для деякого типу. Пригадайте ім'я **PRODUCT** з прикладу 5 попередньої лекції, визначене директивою **typedef** для структури, що описувала товар. Пригадайте також назву типу **fpos_t** – вона також визначена цією директивою як аналог типу **long int**. Загальний синтаксис цієї інструкції такий:

```
typedef <тип> <нове_ім'я>;
```

Приклади

4.
```
//  Визначаємо назву типу WORD_ :
typedef unsigned short WORD_;
WORD_  my_word;  // змінна my_word має тип unsigned short
```

З допомогою інструкції **typedef** можна визначити і тип вказівника на функцію, наприклад таким чином:

```
typedef double (*p_function) (double);
```

Тепер **p_function** – це ***назва типу вказівника на функцію***, яка приймає аргумент типу **double** та повертає результат типу **double**. Цьому вказівнику можна присвоїти ідентифікатор будь-якої функції відповідного виду (ідентифікатор функції є вказівником на неї) або ***передати у функцію в якості параметра***. Щоб оцінити останню можливість, а саме – ім'я функції може бути передано в якості параметру в іншу функцію (sic!), розглянемо приклад чисельного обчислення інтегралу по відрізку від деякої функції. Остання буде параметром функції обчислення інтегралу, і це дасть нам можливість просто міняти імена функцій у виклику, не змінюючи код самої функції.

```
#include <iostream>
#include <cmath>
using namespace std;
// визначаємо тип під іменем p_fun - вказівник на функцію
typedef double (*p_fun) (double);
// підінтегральна функція (її можна задати довільно):
double f (double x);
//Функція int_eps обчислює значення інтеграла із заданою точністю
// eps, де 0<eps<1 від підінтегральної функції, заданої вказівником p
// Параметри функції:     [a, b] – проміжок інтегрування,
//                        eps – точність обчислення
//                        p – вказівник типу p_fun на підінтегральну
```

```
//                          функцію
double int_eps (double a, double b, double eps, p_fun p);
// Функція int_n обчислює значення інтеграла для заданої кількості
// інтервалів n від підінтегральної функції, заданої вказівником p
// Параметри функції:     [a, b] – проміжок інтегрування,
//                        n – кількість інтервалів розбиття проміжку
//                        p – вказівник типу p_fun на підінтегральну
//                        функцію
double int_n (double a, double b, int n, p_fun p);
main ()
{
   double a, b, eps;
   p_fun p = f;           // вказівник p зараз вказує на функцію f
// p_fun p = sin;         // вказівник p зараз вказує на функцію sin
   cout << "Input a = " ;      // вводимо проміжок інтегрування
   cin >> a;
   cout << "\nInput b = " ;    // вводимо проміжок інтегрування
   cin >> b;
   cout << "\nInput eps = " ;//вводимо точність обчислення інтегралу
   cin >> eps;
          // друкуємо результат обчислення інтегралу
   cout << "Integral = " << int_eps (a, b, eps, p) << endl;
   return 0;
}
// Ця функція обчислює значення інтеграла для заданої кількості
// інтервалів n розбиття відрізку [a, b] методом трапецій
double int_n (double a, double b, int n, p_fun p)
{
    double xk = a, delta = (b - a)/ n;  // delta – діаметр розбиття
    double s = (p (a) + p (b)) * 0.5;
       for (int i = 1; i <= n-1; i++)
       {
        xk += delta;
        s += p (xk);
       }
   s *= delta;
   return s;
}
// Ця функція обчислює значення інтеграла
// із заданою точністю eps (0<eps<1)
double int_eps (double a, double b, double eps, p_fun p)
{
   int n = 10;
   double s2, s1 = int_n (a, b, n, p);
      do
      {
       s2 = s1;
       n *= 2;
       s1 = int_n (a, b, n, p);
```

```
        }
        while (fabs (s1 - s2) >= eps);
    return s1;
}
// Це підінтегральна функція:
double f (double x)
{
    return x;    // в даному випадку – це просто пряма: y = x
}
```

Зауваження. В цьому прикладі використовується функція **int_eps**, яка для заданого відрізку **[a, b]** обчислює наближене з точністю **eps** значення інтегралу від підінтегральної функції, вказаної вказівником **p**, яка передається як параметр. Алгоритм обчислення зовсім простий – спочатку відрізок **[a,b]** ділиться на 10 інтервалів, викликається функція **int_n**, яка методом трапецій обчислює наближене значення інтегралу, потім кількість інтервалів розбиття подвоюється, функція **int_n** обчислює ще одне наближене значення інтегралу. Якщо модуль різниці наближень менше заданої точності **eps**, то останнє наближення є остаточним результатом. Якщо ж ні, то процедура повторюється. Щоб змінити підінтегральну функцію, досить у функції **main** встановити вказівник **p** на будь-яку системну чи власну функцію з єдиним обмеженням: вона повинна мати такий самий тип, який визначений іменем **p_fun**, тобто мати один параметр типу **double** та повертати результат типу **double**.

Контрольні питання.

1. Що таке препроцесор?
2. Що таке макрос?
3. Які особливості макровизначення з параметрами?
4. Яка різниця в директивах **#include <file>** та **#include "file"**?
5. Як реалізувати передачу функції в якості параметру в деяку функцію. Придумайте ситуацію, коли це може бути корисним (окрім наближеного обчислення значення інтегралу)

Завдання для самоперевірки.

1. Написати макровизначення, яке визначає квадрат свого аргументу та переконайтесь у правильності його роботи.
2. Написати макровизначення, яке визначає максимум з трьох своїх аргументів.

Лекція 9. Деякі нові можливості, які надає мова C++ в порівнянні з класичною мовою C.

1. Вбудовані функції.

В C++ введені в обіг так звані вбудовані функції (позначаються службовим словом **inline** у визначенні функції). Ідея їх використання досить проста – невелика за обсягом коду функція просто «вбудовується» в код в місцях виклику. Виграш очевидний – економія часу та ресурсів на виклик функції, недолік теж неприхований – при великій кількості звертань до такої функції код програми розростається. Перевага перед макросами теж неабияка – всі побічні ефекти, пов'язані із викликами, знімаються. Вбудована функція має бути визначена (декларації в цьому разі замало) перед першим звертанням до неї.

Слід зауважити, втім, що не всяка функція може бути вбудованою. Наприклад, для рекурсивної функції це, вочевидь, неможливо. Інші обмеження може накладати компілятор – функція, в якій є цикли або розгалуження також може не сприйматись компілятором як **inline** –функція. Проте при нагоді старайтесь використовувати цю можливість С++.

Приклади

1.
```
// порівнюємо макровизначення та inline-функцію
#include <iostream>
#define cube(x) ((x)*(x)*(x))        // макрос з параметром
inline double cube_ (double x)       // вбудована функція
{
     return x*x*x;
}
using namespace std;
int main ()
{
     int i = 1;
     cout << cube (++i) << " " << i << endl;
     i = 1;
     cout << cube_ (++i) << " " << i << endl;
     return 0;
}
```

Зауваження. Раніше ми вже обговорювали, що подібне звертання до макросу: **cube (++i)** призводить до невірного результату. Крім того, змінна **i** буде інкрементована тричі. А от результат виклику вбудованої функції цілком правильний. Переконайтесь, що результат виконання цього прикладу буде таким:

```
36 4
8 2
```

2. Змінні типу посилання (reference variables)

Всі типи вказівників, прийняті в С, очевидно підтримуються і в С++ в силу спадкоємності. Розширенням С++ є так звані ***змінні типу посилання***. Ідея полягає в тому, що в момент визначення такої змінної до її ідентифікатора приписується знак **&** (амперсанд, який позначає взяття адреси), а після знаку **=** вказується ідентифікатор тієї змінної, на яку і буде посилатись дана **reference**-змінна. Наприклад:

Приклади

2.
```
// знайомимось з reference-змінною
#include <iostream>
using namespace std;
int main ()
{
     int i = 1;
     int &r_i = i;  // Змінна-посилання r_i є псевдонімом змінної i
     int *pi = &i;  // Визначаємо вказівник, який вказує на змінну i
     // наступний вивід на екран показує, що i, r_i та *pi - однакові
     cout << i << " " << r_i << " " << *pi << " " << pi << endl;
     r_i = 5;       // Змінній  r_i присвоєне значення 5
     // наступний вивід на екран показує, що i, r_i та *pi - однакові
```

```
    cout << i << " " << r_i << " " << *pi << " " << pi << endl;
    *pi = 10; // Значення 10 присвоєно змінній,
              // на яку вказує вказівник pi,
    // наступний вивід на екран показує, що i, r_i та *pi - однакові
    cout << i << " " << r_i << " " << *pi << " " << pi << endl;
    pi  = &r_i;  // вказівник pi одержує адресу reference-змінної
    // наступний вивід на екран показує, що i, r_i та *pi - однакові
    cout << i << " " << r_i << " " << *pi << " " << pi << endl;
    return 0;
}
```

Зауваження. Результат виконання цього прикладу такий:

```
1 1 1 0x6ffdfc
5 5 5 0x6ffdfc
10 10 10 0x6ffdfc
10 10 10 0x6ffdfc
```

Він свідчить, що змінні **i** та **r_i** – це суть одна й та сама змінна, тому і адреси їх співпадають.

Таким чином, змінна типу посилання є з одного боку псевдонімом деякої змінної, розташованої в пам'яті, а з іншого боку – поводить себе, як посилання на цю змінну, з усіма наслідками, які випливають, наприклад, у зв'язку з передачею параметрів в функціях.

Власне, передача параметрів у функцію через механізм **reference**-змінних, це і є їх основне призначення. Пригадаємо приклади 2А та 2Б з лекції 5. В них ми обговорили різницю у механізмах передачі параметрів за значенням та через вказівник. Зараз створимо ще один варіант цієї самої функції, але з передачею параметрів через **reference**-змінні, або, що те саме, ***за посиланням***.

Приклади

3.
```
    // Функція change – передача параметрів by-reference
    // (за посиланням)
#include <iostream>
using namespace std;
void change (int &a, int &b)  // Параметри a та b передаються
                              // через посилання
{
   int   c;
   c = a;
   a = b;
   b = c;
   cout << "\n In change: " << a << " " << b << endl;
}
main ()
{
int   x = 1, y = 2;
   cout << "\n main (before change): " << x << " " << y << endl;
   change (x, y);
   cout << "\n main (after change): " << x << " " << y << endl;
return 0;
}
```

Зауваження. Зверніть увагу: при виклику даної функції **change** формальні параметри цієї функції **a** та **b** стають посиланнями (псевдонімами) фактичних аргументів **x** та **y**. Тому всі зміни, які запрограмовані для змінних **a** та **b** всередині функції, реально відбуваються із **x** та **y**. В даному випадку ми одержали саме той результат, для якого призначена ця функція. (І ніяких вказівників!) Давайте все ж таки визнаємо, що знайомство із вказівниками, якщо і лишило важкі спогади, тим не менше, безумовно, було корисним, особливо приймаючи до уваги, що і надалі ми будемо їх використовувати.
Результат виконання цього прикладу такий:

```
main (before change): 1 2
In change: 2 1
main (after change): 2 1
```

Коли має сенс використовувати передачу параметрів by-reference (за посиланням):

- якщо необхідно, щоб функція змінювала аргументи, які передає їй функція, що її викликає;
- якщо треба підвищити швидкодію програми – за рахунок передачі функції посилань замість об'єктів даних (не витрачається час на створення копій у стеку)

З іншого боку, слід розуміти, що такий спосіб передачі параметрів у функцію, робить ці дані вразливими для випадкових і небажаних змін.

Як передати параметри у функцію, щоб вона не змінювала передані їй у виклику аргументи :

- якщо розміри об'єкту даних невеликі, наприклад це вбудований тип даних або невелика структура – варто передавати його за значенням;
- якщо об'єктом даних є масив – використовуйте вказівник (він передається за значенням), проте для захисту даних задекларуйте його як вказівник на **const**;
- якщо об'єктом даних є велика структура – використовуйте вказівник на **const** або **const**-посилання для економії часу та пам'яті (об'єкт не копіюватиметься у стек);
- якщо об'єктом даних є екземпляр класу – використовуйте **const**-посилання для економії часу та пам'яті .

Як передати параметри у функцію, щоб вона могла змінити передані їй у виклику аргументи :

- якщо об'єктом даних є вбудований тип – використовуйте вказівник;
- якщо об'єктом даних є масив – єдиним правильним варіантом є вказівник;
- якщо об'єктом даних є структура – використовуйте вказівник або посилання;
- якщо об'єктом даних є екземпляр класу – використовуйте посилання.

Тепер, коли в нашому розпорядженні 3 способи передачі параметрів у функції, варто вміти чітко розмежовувати ситуації використання того чи іншого способу.

3. Перевантаження функцій – простий поліморфізм.

Один із суттєвих аспектів мови C++, який принципово відрізняє її від C, – це ***перевантаження імен функцій***. Перевантаження дозволяє мати декілька функцій *з* ***однаковими ідентифікаторами*** в одній і тій самій області видимості, які виконують операції над аргументами ***різних типів***. Насправді це дуже складне питання, адже компілятор розрізняє функції саме по їх іменах, а тут приходиться розв'язувати питання про те, яка саме функція має бути викликана лише в залежності від типів фактичних параметрів.

Питання про корисність таких функцій має очевидну відповідь. Уявіть собі, що необхідно вміти обчислювати куб деякої змінної, яка може бути цілою, дійсною або подвоєної точності. В цьому разі при загальному підході треба створювати декілька різних функцій (з різними, звісно, іменами) і не заплутатись при їх викликах. Перевантаження функцій як раз і дозволяє в цій ситуації використовувати різні функції, але з одним і тим самим іменем, що цілком природно. Наприклад,

```
float cube ( float v )
{
   return v*v*v;
}
int cube ( int v )
{
   return v*v*v;
}
double cube ( double v )
{
   return v*v*v;
}
int main (void)
{
int    i = 1;
float  f = 2.5;
double d = 1.5e-10;
   i = cube (i);  // Тут викликається  cube ( int )
   f = cube (f);  // Тут викликається  cube ( float )
   d = cube (d);  // Тут викликається  cube ( double )
}
```

Як технічно здійснюється перевантаження? Це відбувається за рахунок спеціальної операції **name mangling** – спотворення імені, зміст якого полягає в тому, що компілятор до імені кожної функції приєднує деякі символи, які визначаються типами параметрів даної функції. Тому важливо, щоб параметри функцій, які перевантажуються, були ***досить різними*** – за кількістю або за типами, тобто функції мають розрізнятись своїми ***сигнатурами*** (до сигнатури функції входять її ідентифікатор та типи параметрів, але не результат)

4. Умовчання для параметрів функцій.

Мається на увазі можливість задавати значення за умовчанням деяким параметрам функцій. Таким чином при виклику відповідні аргументи, якщо вони пропущені, будуть замінені значеннями за умовчанням.

Приклад.

```
// Припустимо, що це функція ініціалізації текстового режиму
// на екрані
void InitScreen (int width = 80,
                 int height = 24,
                 char background = '  ' )
{    // код функції }
int main () {
// Можливі виклики:
 InitScreen (60);
```

```
 InitScreen (60, 20);
 InitScreen (60, 20, '#');
// Неможливо використати умовчання лише для
// останнього параметру у виклику :
 InitScreen ('*');
}
```

Контрольні питання.

1. Як визначається та ініціалізується змінна типу посилання?
2. Які способи передачі параметрів у функцію вам відомі? Коли і який спосіб варто використовувати?
3. В чому полягає особливість вбудованих функцій? Чи будь-яка функція може бути оголошена вбудованою?
4. Що таке перевантажені функції, як реалізований механізм перевантаження імен функцій?
5. Як визначаються аргументи функцій за замовченням?

Завдання для самопідготовки.

1. Визначити дві перевантажені функції, які обчислюють мінімум з двох своїх дійсних або символьних параметрів.
2. Визначити для цих функцій аргументи за замовченням.

Лекція 10. Клас в С++.

1. Поняття про клас в С++.

Механізм класів мови С++ призначений для створення власних типів даних з метою реалізації абстракцій реальних об'єктів та співвідношень між ними.

Основою для введення нового поняття в мову С++ були структури класичної мови С. Наприклад, для реалізації математичного поняття комплексного числа в лекції 6 ми використовували структуру:

```
struct COMPLEX {float Re, Im;};     // Визначення типу структури
COMPLEX   A, B;      // Визначення комплексних змінних A, B
```

Але операції над комплексними числами (додавання, множення, ділення) в межах мови С могли бути реалізовані лише у вигляді функцій, які використовують структури у вигляді параметрів.

На відміну від структур клас не лише містить пасивні дані-члени, а і має засоби для їх активного використання – функції-члени. Визначення класу складається із заголовку класу та тіла класу, поміщеного в фігурні дужки. Наприклад:

```
class Example   //  Це заголовок класу з тегом (іменем) Example
{
                //  Це тіло класу
};
```

Будемо називати змінні, які відносяться до даного типу класу, ***екземплярами класу***.

```
Example myExample;  // Визначаємо екземпляр myExample класу
                    // Example
```

Зауваження. Тіло класу визначає окрему область видимості. Отже, якщо в різних класах є члени з однаковими іменами, то вони відносяться до різних областей видимості і не

конфліктують між собою. В межах одного класу всі члени класу, очевидно, мають унікальні імена.

В тілі класу дані-члени та функції-члени класу визначаються із вказанням рівнів доступу до них (поки що обговоримо лише два з них **public** – відкритий (загально доступний) та **private** – закритий, приватний). Визначення даних-членів мають такий самий синтаксис, як і визначення членів структур. Функції-члени можуть визначатись як всередині класу (***тоді за замовченням вони вважаються вбудованими (inline) функціями***), так і поза ним. Але їх ***декларації обов'язково знаходяться в класі***.

Для прикладу розглянемо визначення класу, який містить інформацію про банківський рахунок та виконує деякі дії з цією інформацією – дає можливість поповнити рахунок, зняти суму з рахунку та визначити поточний баланс грошей на рахунку. Вказані дії будуть реалізовані з допомогою функцій-членів класу.

Приклади

1.
```
// Створюємо клас Account – банківський рахунок:
class Account              //  Це заголовок класу Account
{
   private:                //  це секція закритих членів класу
                  //  далі визначаються дані-члени (або поля класу)
     string name;          //  ім'я господаря рахунку
     int  number;          //  номер рахунку
     double money;         //  сума на рахунку
     int pin;              // пін-код доступу до рахунку

   public:                 //  це секція відкритих членів класу
                  //  далі декларуються або визначаються функції-члени
                  // (або методи класу)
     void add_money (double sum)   // поповнення рахунку
     {
          if (sum > 0) money += sum;    // додаємо вказану суму sum
     }
     double my_money (int my_pin)  // повертається сума на рахунку
     {
          if (my_pin == pin) return money; // якщо вірний pin, то
     // повертається поточна сума на рахунку
     }

     // ця функція лише декларується, а визначена буде поза класом
     void get_money (int my_pin, double sum); // зняття готівки
};
```

Для того, щоб визначити функцію-член класу **get_money** (вона в класі була лише продекларована), необхідно обов'язково вказати, що вона є членом саме класу **Account**. Для цього використовується оператор області видимості **::** , який прив'язує певний ідентифікатор, до деякої конкретної області видимості. В даному випадку за межами класу повний ідентифікатор цієї функції буде **Account :: get_money**.

Логіка цієї функція проста – якщо вказаний правильний пін-код і сума зняття додатна, проте не перевищує наявної суми на рахунку, то – знімаємо її, тобто зменшуємо відповідним чином суму на рахунку.

```
void Account :: get_money (int my_pin, double sum)
```

```
{
      if ((my_pin == pin)&&(sum > 0)&&(sum <= money)) money -= sum;
}
```

Для створення екземпляру класу **Account** необхідно помістити у функції **main** визначення: **Account Me;** або визначити вказівник на екземпляр класу : **Account* pA**, і потім створити новий екземпляр, що адресується цим вказівником, в області **Heap** за допомогою оператору **new:**

```
pA = new Account;    // створений екземпляр *pA
```

Доступ до членів класу здійснюється так само, як і доступ до членів структур – через операцію **.** (операція крапка), або, якщо екземпляр адресується вказівником, через операцію **->**.

В кінці роботи екземпляр, створений динамічно оператором **new**, треба знищити оператором **delete**:

```
delete pA;
```

Функції-члени класу можуть перевантажуватись, як і будь-які інші функції та мати значення параметрів за замовчуванням.

Множина відкритих членів класу називається ***інтерфейсом*** класу.

Службове слово **private** означає, що члени класу, які мають даний статус, можуть бути використані лише в межах даного класу і є недоступними для жодної зовнішньої відносно класу функції. Метою цього є приховати, зберегти цілісність інформації в класі. Як правило, статус **private** мають дані-члени. В такому разі змінювати їх можуть лише функції-члени даного класу, лише вони можуть повертати значення закритих членів класу або контрольовано змінювати їх. Ця стратегія носить назву ***інкапсуляція*** і є однією з складових концепції об'єктно-орієнтованого програмування (перегляньте обговорення цього принципу у вступі до даного посібника).

Слід зауважити, що деякі зовнішні функції теж мають доступ до закритих членів класу. Це так звані ***дружні функції*** класу. Досить помістити їх декларацію в класі зі службовим словом **friend**, наприклад, можна зробити другом класу таку функцію (вона повертає пін-код переданого їй рахунку!):

```
      int I_take_your_pin (const Account &rA) {return rA.pin;}

class Account              //  Це заголовок класу Account
{
      … // те, що раніше
      friend I_take_your_pin (const Account &rA); // дружня функція
};
```

то доля пін-коду вашого рахунку невідома. Обов'язково спробуйте викликати цю функцію, але вилучивши з класу її декларацію із словом **friend**. Який результат ви одержите?

Зверніть увагу – клас може бути параметром функції і передаватись за значенням, за посиланням, або через вказівник. Конкретно в даній дружній функції екземпляр класу передається за посиланням, проте специфікатор **const** не дає права функції змінювати зміст екземпляру. Проте наразі все ще незрозуміло, яким чином можна задавати значення закритих полів класу. Адже створивши конкретний екземпляр рахунку, нам необхідно ввести в нього початкову інформацію. Звісно, можна написати функцію-член класу, яка буде призначена для ініціалізації екземпляру і викликати її при створенні екземпляру. Проте, звісно, зручніше, щоб подібні дії відбувались при створенні екземпляру автоматично – за рахунок виклику подібної

функції компілятором. Це означає, що компілятор має «впізнати» цю функцію-ініціалізатор серед всіх функцій-членів класу. Так був знайдений геніальний вихід – така функція, яка має автоматично викликатись компілятором при створенні екземпляру, має мати ту ж саму назву, що й клас!

2. Конструктор та деструктор класу.

Отже, функція, необхідність якої була щойно з'ясована, називається ***конструктором класу***. Вона має бути відкритою, носити назву класу і за синтаксисом не мати типу результату, навіть **void**.

Як і інші функції-члени класу конструктор може перевантажуватись. В класі може не бути визначений конструктор, в такому разі буде автоматично створений так званий «конструктор за умовчанням» (**by default**). Подібний конструктор проініціалізує всі поля екземпляру нульовими значеннями. Якщо ж в класі визначений хоч один конструктор, то подібного конструктора за умовчанням, що не має параметрів, створено не буде.

Конструктор викликається ***тільки автоматично***, це виключно прерогатива компілятора.

Оскільки всі ресурси, які задіяв клас, – наприклад відкриті файли, виділена пам'ять, тощо, мають бути звільнені в момент знищення екземпляру класу, (локальні екземпляри знищуються в момент завершення функцією роботи, глобальні – в кінці програми), то за це теж повинна нести відповідальність певна функція-член класу. Вона буде викликатися автоматично в порядку, зворотному порядку виникнення екземплярів. Ця функція-член класу називається ***деструктор***. Її ім'я – це ім'я класу із знаком **~** (тильда – заперечення).

Деструктор не має типу результату, як і конструктор, і не може мати параметрів, отже, не може перевантажуватись. Так само, як і конструктор, в разі відсутності деструктора в класі, буде створений деструктор за умовчанням.

В нашому класі **Account** конструктор необхідний, а от деструктор ми включимо в клас просто із повчальною метою. (Дружню функцію, про яку йшлося вище, навпаки – в клас включати не будемо. Перевірте її використання самостійно.)

Приклади

2.
```
// Остаточний вигляд класу  Account:
#include <iostream>
#include <string>
using namespace std;

class Account              //  Це заголовок класу Account
{
   private:                //  це секція закритих членів класу
              //  далі визначаються дані-члени (або поля класу)
     string name;          //  ім'я господаря рахунку
     int  number;          //  номер рахунку
     double money;         //  сума на рахунку
     int pin;              // пін-код доступу до рахунку

   public:                 //  це секція відкритих членів класу
             //  далі визначаються та декларуються функції-члени
             //(або методи класу
```

```
      // декларація конструктора класу –
      // для останнього параметра визначено умовчання
      Account (string my_name, int my_number, int pin,
                double my_money = 0);
      ~Account ()     // деструктор, тут він "декоративний"
      {
            cout << "Delete Account " << name << endl;
      }
      void add_money (double sum)    //функція для поповнення рахунку
      {
            if (sum > 0) money += sum;
      }
      double my_money (int my_pin)//функція повертає суму на рахунку
      {
            if (my_pin == pin) return money;
      }
// ця функція лише декларується, а визначена буде поза класом

     void get_money (int my_pin, double sum);// функція зняття готівки
};    // кінець визначення класу, крапка з комою необхідна

// визначення конструктора поза класом
Account :: Account  (string my_name, int my_number, int my_pin,
                     double my_money)
{
      number = my_number;
      pin = my_pin;
      money = my_money;
      name = my_name;
      cout << "Create an Account " << name << endl;
}

// визначення функції get_money поза класом
void Account :: get_money (int my_pin, double sum)
{
      if ((my_pin == pin)&&(sum > 0)&&(sum <= money)) money -= sum;
}

int main()
{
      // Створюємо екземпляр Me класу Account
      Account Me ("Petrenko_Petro", 123456, 1111, 1000);
      // Виводимо для нього поточний баланс
      cout << "1) Balance: " << Me.my_money(1111) << endl;
      // Створюємо динамічний екземпляр через вказівник
      // За умовчанням баланс, тобто money = 0
      Account *pA = new Account ("Ivanenko_Ivan", 123457, 2222);
      pA->add_money(2000);       // викликаємо функцію add_money –
      // поповнюємо рахунок на 2000
```

```
    pA->get_money(2222, 500); // викликаємо функцію get_money –
    // знімаємо 500
    // Виводимо для цього екземпляра поточний баланс:
    cout << "2) Balance: " << pA->my_money(2222) << endl;
    delete pA;      // Знищили динамічний екземпляр, тобто звільнили
    // пам'ять

}
```

Зауваження.

Результат виконання даного прикладу буде таким:

```
Create an Account Petrenko_Petro
1) Balance: 1000
Create an Account Ivanenko_Ivan
2) Balance: 1500
Delete Account Ivanenko_Ivan
Delete Account Petrenko_Petro
```

Зверніть увагу, перше повідомлення створює конструктор при створенні статичного екземпляра **Me**:

```
Account Me ("Petrenko_Petro", 123456, 1111, 1000);
```

Далі виводиться баланс рахунку для даного екземпляру – це результат виклику функції **Me.my_money(1111)**. Наступне повідомлення знову є результатом роботи конструктора, але тепер для динамічного екземпляра, який адресується вказівником **pA**. Для створення цього динамічного екземпляра конструктор використав умовчання для останнього параметра **my_money = 0**, тому викликом функції **add_money(2000)** поповнили даний рахунок, потім зняли з нього певну суму через функцію **get_money** та вивели на екран поточний баланс, який, як неважко перевірити, дійсно становить **1500**. Далі бачимо повідомлення про знищення динамічно створеного екземпляру рахунку – це результат виконання команди **delete pA;**, яка приводить до автоматичного виклику конструктора. І нарешті, статичний екземпляр **Me** буде знищений при завершенні роботи функції **main**, про що нас повідомить деструктор.

3. Створення проекту в середовищі DEV-CPP.

Вдосконалимо попередній приклад, розмістивши фрагменти коду в різних файлах згідно прийнятому стилю ООП. Визначення класу помістимо у файл з розширенням **.h** (наприклад, **account.h**). При цьому використаємо команду препроцесора **#ifndef**, щоб позбутись повторного включення коду. Визначення функцій класу помістимо у файл з тим самим іменем, що й клас, і розширенням **.cpp** (**account.cpp**) і нарешті код, який використовує даний клас, тобто клієнтський файл, – у файл **use_account.cpp**. Для запуску проекту використовуйте файл проекту з розширенням **dev**.

Алгоритм створення проекту в середовищі **DEV-CPP**:

1. Створити окрему папку, в яку будуть поміщатись всі файли проекту.
2. Обрати пункт меню Файл→Створити→Проект (створюється файл main.cpp – зберігти його під потрібним іменем, наприклад, **use_account.cpp** і помістити в нього код використання класу).
3. Тепер обрати пункт меню Файл→Створити→Вихідний файл (на питання "Додати файл в проект?" дати згоду). Зберігти файл під потрібним іменем (наприклад, **account.h**) і

помістити в нього визначення класу **Account**. На початку файлу включити директиви **#ifndef ACCOUNT_H** та **#define ACCOUNT_H**. В кінці файлу включити директиву **#endif**

4. Знову обрати пункт меню Файл→Створити→Вихідний файл (на питання "Додати файл в проект?" дати згоду). Зберігти файл під потрібним іменем (наприклад, **account.cpp**) і помістити в нього визначення функцій класу. На початку файлу не забути про директиву **#include "account.h"**. Така сама директива має бути і в файлі **use_account.cpp.**

Контрольні питання.

1. Що таке клас, який синтаксис його визначення?
2. Що таке екземпляр класу, як йог створити?
3. Що таке конструктор класу, коли та як він буде викликаний?
4. Що таке деструктор класу, коли та як він буде викликаний?
5. Яким чином визначаються функції-члени класу поза тілом класу?
6. Як визначити дружню функцію, яка її роль?

Завдання для самопідготовки.

1. Відпрацювати завдання щодо дружньої функції, яке було сформульоване в тексті.
2. Створити клас **Student**, який містить інформацію про студента – його ім'я, курс, та середню семестрову і середню екзаменаційну оцінки. Створіть в класі конструктор, передбачте в разі необхідності умовчання до деяких його параметрів. Створіть статичний та динамічний екземпляри **Student**. Переконайтесь, що ви розумієте, коли та як був викликаний конструктор.
3. Повторіть попереднє завдання, але створивши окремий проект з header-файлом, файлом визначення функцій-членів класу та файлом, що містить функцію **main**.

Лекція 11. Перевантаження операцій в мові C++.

Пригадаємо, що C++ – типізована мова високого рівня. Це означає, що будь-який об'єкт в програмі має певний тип, а відтак і визначає множину допустимих операцій над об'єктами даного типу. Пригадайте, саме типами операндів та типом результату визначався поділ всіх операцій мов C/C++ на певні групи. Оскільки зараз ми опанували можливість створення об'єктів нових користувацьких типів (класів), то виникає певна необхідність виконувати деякі дії з об'єктами класів, особливо, якщо вони реалізують певні математичні поняття, для яких як раз і визначені математичні операції. Наприклад, об'єкти класу, який реалізує геометричний вектор, варто вміти додавати, множити на число, визначати скалярний добуток, тощо. Одним із шляхів реалізації операцій над об'єктами класів є створення набору методів, які їх будуть виконувати. Але, погодьтесь, значно логічніше виглядало би створення екземплярів згаданого вище класу геометричних векторів і виконання операцій таким чином:

```
vector a, b, c;
c = a + b;
```

Тобто, хотілось би використовувати звичні знаки операцій для їх реалізації замість виклику методів. Це означає, що потрібно «навчити» компілятор та класи розуміти ці знаки операцій і у відповідності до них, передавати їх на виконання відповідним функціям-членам класу. Ця технологія називається ***перевантаженням операцій*** в класі.

1. Загальні правила перевантаження операцій.

Для перевантаження більшості операцій в мові C++ закладені дві можливості: використання так званих операторних методів, які є функціями-членами даного класу або використання операторних функцій, які є «друзями» класу. У будь-якому випадку стає зрозумілим, що ці операторні функції повинні мати особливі ідентифікатори. По-перше, їх повинен «впізнати» компілятор, по-друге, має бути зрозумілим, якій саме операції відповідає конкретна операторна функція. Ця проблема реалізована таким чином: операторна функція, призначена для реалізації операції, яка позначається умовним знаком **#**, називається **operator#**. Подальші нюанси пов'язані із різницею у перевантаженні унарних та бінарних операцій, про що буде сказано далі. Слід зауважити відразу ряд обмежень, які діють при перевантаженні операцій – неможливо вигадати свої знаки операцій, допускається використання лише визначених в мові знаків операцій. По-друге, неможливо змінити пріоритет перевантаженої операції, і по-третє, унарні операції мають перевантажуватись як унарні, а бінарні – відповідно, як бінарні. Окрім того, діє заборона на перевантаження деяких операцій, зокрема не перевантажуються такі операції:

sizeof (визначення об'єму пам'яті обєкта)

. (доступ до елементу)

***** (операція розадресації вказівника на елемент)

:: (оператор області видимості)

?: (тернарна операція)

та деякі інші, пов'язані з приведенням та перетворенням типів.

В багатьох випадках не має різниці, яку саме форму операторної функції (функція-член класу чи дружня функція) ви використовуєте. Проте, якщо ви намагаєтесь перевантажити деяку операцію, яка буде використовуватись у виразах, де перший операнд не є екземпляром даного класу (скажімо, при множенні скаляра на екземпляр класу, що реалізує вектор або матрицю), то операторна функція обов'язково має бути реалізована як дружня функція.

З іншого боку, деякі операції можуть перевантажуватись лише з допомогою функцій-членів класу. Це такі операції:

= (присвоєння)

() (оператор виклику функції)

[] (оператор індексації)

-> (доступ до членів класу через вказівник).

2. Вказівник this.

Кожна функція-член класу (в тому числі конструктори та деструктор) неявним параметром мають особливий вказівник **this**. Характерною особливістю його є те, що він вказує на поточний екземпляр, тобто той, для якого здійснюється виклик даної функції. Наприклад, у прикладі 2 з попередньої лекції при виклику функції **Me.my_money(…)** вказівник **this** вказує на екземпляр **Me** (тобто є його адресою), при виклику функції **pA->add_money(…)** значення вказівника **this** збігається із значенням вказівника **pA**. Вказівник **this** також може використовуватись безпосередньо у будь-якій з функцій класу. Наприклад, визначення функції **my_money** у згаданому прикладі могло би виглядати таким чином:

```
double my_money (int my_pin)  // сума на рахунку
{
```

```
        if (my_pin == this->pin) return this->money;
    }
```

Конкретно у даному випадку необхідності у використанні вказівника **this** немає, проте саме так доповнює компілятор звертання до всіх членів класу.

3. Перевантаження бінарних операцій.

Якщо ми перевантажуємо бінарну операції **#** з допомогою дружньої для класу функції, тобто в коді програми цією функцією має бути реалізоване виконання виразу **a#b**, то синтаксис декларації цієї дружньої операторної функції в класі має бути таким:

```
friend <тип_результату> operator#
        (<тип_операнду_1> op1, < тип_опера@нду_2> op2);
```

При цьому вираз виду **a#b** буде замінений компілятором на такий виклик дружньої операторної функції: **operator# (a,b).**

Уявімо собі, що в класі геометричних векторів **vector** ми реалізуємо виконання операції додавання саме в такий спосіб, тоді в класі ми декларуємо дружню функцію

```
friend vector operator+ (vector a, vector b);
```

А в самій функції прописуємо, яким чином відбувається додавання двох векторів **a** та **b.** В результаті в коді вираз виду **a + b**. буде автоматично замінений на виклик функції **operator+ (a, b)**. (Нагадаємо, що саме так перевантажуються операції в класах мови C#, проте там не використовуються дружні функції, є лише функції-члени класу, які мають саме такий синтаксис визначення).

Дещо інакше виглядає ситуація, коли перевантаження бінарної операції **#** відбувається з допомогою функції-члену класу. Тоді перший операнд операції передається у цю операторну функцію не як параметр, а неявно у вигляді вказівника **this** на поточний екземпляр класу. Тобто параметром такої операторної функції є лише другий операнд операції. Це означає, що синтаксис декларації операторної функції-члену класу, що перевантажує операцію **#** такий:

```
<тип_результату> operator# (< тип_опера@нду_2> op2);
```

а вираз виду **a#b** буде замінений компілятором на такий виклик операторної функції-члену класу: **a.operator# (b).**

Звідси стає зрозумілим, чому функція член класу не може перевантажити бінарну операцію у випадку, коли її перший операнд має тип, відмінний від типу класу.

Повертаючись до того самого уявного прикладу із додаванням геометричних векторів, розуміємо, що функція-член класу повинна бути продекларована так:

```
vector operator+ (vector b);
```

При цьому вираз **a + b** в коді програми буде замінений на виклик: **a.operator+ (b)** .

4. Перевантаження унарних операцій.

Якщо унарна операція **#** перевантажується з допомогою дружньої для класу функції, тобто в коді програми цією функцією має бути реалізоване виконання виразу **#a**, отже, синтаксис декларації цієї функції в класі має бути таким:

```
friend <тип_результату> operator# (<тип_операнду_1> op1);
```

При цьому вираз виду **#a** буде замінений компілятором на такий виклик дружньої операторної функції: **operator# (a).**

Уявімо собі, що в класі геометричних векторів **vector** ми реалізуємо виконання операції переходу від вектора **a** до протилежного вектора **-a** дружньою функцією, тоді в клас **vector** ми поміщаємо таку декларацію дружньої функції

```
friend vector operator- (vector a);
```
В самій функції маємо визначити, яким чином створюється протилежний вектор до **a.** В результаті в коді вираз виду **-a** буде автоматично замінений на виклик функції **operator- (a).**

Якщо ж перевантаження унарної операції **#** відбувається з допомогою функції-члену класу, то єдиний операнд цієї операції потрапляє у цю операторну функцію у вигляді вказівника **this** на поточний екземпляр класу. Тобто параметри у такої операторної функції відсутні. Це означає, що синтаксис декларації операторної функції, що перевантажує операцію **#** такий:

```
<тип_результату> operator# ( );
```
а вираз виду **#a** операторної функції-члену класу: **a.operator# ( ).**

Для кращого уявлення вкажемо, що функція член класу **vector**, яка реалізує перехід до протилежного вектору буде продекларована таким чином:

```
vector operator- ( );
```
При цьому вираз **-a** в коді програми буде замінений на виклик: **a.operator- ( ).**

Ще один нюанс визначення унарних операцій пов'язаний із операціями інкременту та декременту. Як відомо, і інкремент, і декремент можуть мати префіксну або постфіксну форми (до речі, в мові C# вони не розрізняються, перевантажується лише префіксна форма). Мова C++ надає можливість реалізувати префіксні та постфіксні операії, як і належно, по-різному. Для цього в операторній функції, що реалізує постфіксні операції, використовується фіктивний параметр, він ніяк не використовується, лише розрізняє сигнатури перевантажених різних форм інкременту. Отже, для перевантаження цих двох форм інкременту використовуються функції-члени класу із відповідно такими деклараціями:

```
// префіксний інкремент:
<тип_результату> operator++ ();
// постфіксний інкремент:
<тип_результату> operator++ (int unused);
```
Або, якщо операції визначаються з допомогою дружніх функцій, то синтаксис їх декларацій такий:

```
// префіксний інкремент:
friend <тип_результату> operator++ (<тип_операнду>);
// постфіксний інкремент:
friend <тип_результату> operator++
                    (<тип_операнду>, int unused);
```
Абсолютно аналогічні правила діють і для операції декременту.

5. Особливості перевантаження операції присвоєння. Конструктор копіювання.

Слід зазначити, що за умовчанням операція присвоєння (**=**) для двох екземплярів класу здійснює поелементне копіювання даних-членів цих екземплярів. Якщо це саме те, що потрібно для вашого класу, ***немає необхідності у перевантаженні операції присвоєння***.

Проте, якщо членом класу є вказівник або посилання, то за умовчанням відбуватиметься ***копіювання відповідних вказівників чи посилань***, а не об'єктів, на які вони посилаються (чого, скоріше за все, ви очікуєте при присвоєнні). Подібне копіювання, яке здійснюється за умовчанням, носить назву ***поверхневого копіювання***. Якщо ж потрібно від вказівників дістатись до пов'язаних із ними об'єктів, має здійснюватися так зване ***глибоке копіювання***. В

такому разі необхідне коректне перевантаження операції присвоєння (**=**) . Як правило, тоді виникає і потреба в особливому конструкторі – ***конструкторі копіювання***.

Спробуємо з'ясувати його необхідність на прикладі. Для цього повернемось до класу **Account**, створеного у попередній лекції. Спробуйте додати перед функцією **main** ще одну функцію виду: **void fun (Account A) { }**. Вона, як бачимо, не виконує нічого, лише приймає (за значенням) екземпляр **A** класу **Account**. Додайте тепер в самій функції **main** виклик **fun (*pA);** , де вказівник **pA** був визначений як **Account *pA**. Запустіть цей приклад на виконання і перевірте, що ви раптом одержите ***зайве*** повідомлення від деструктора: **Delete Account Ivanenko_Ivan**. Справа полягає в тому, що при передачі об'єкту **A** за значенням, в стеку має бути створена його копія. От цю копію як раз і створює конструктор копіювання. Оскільки в нашому класі **Account** подібного конструктора створено не було, компілятор надав для класу системний конструктор копіювання, який здійснює просте поелементне (***поверхневе***!) копіювання об'єкту. А після завершення виклику функції **fun** деструктор знищив цю копію зі стеку – саме тут з'явилось «зайве» повідомлення від нього. У випадку класу із полями, які не є вказівниками, ніяких проблем не виникає. Проте, поміркуйте, що сталось би, якби ось так деструктор знищив копію посилання на реальний об'єкт?

Задля матеріалізації цих міркувань, розглянемо модельний приклад класу **Example**, в якому є єдине поле – вказівник на змінну цілого типу. Сама ж ця динамічна змінні буде створюватись конструктором і відповідно знищуватись деструктором. І так само, як пропонувалось у попередньому абзаці, використаємо функцію **fun**, яка отримуватиме екземпляр класу за значенням. Ось цей клас.

Приклади

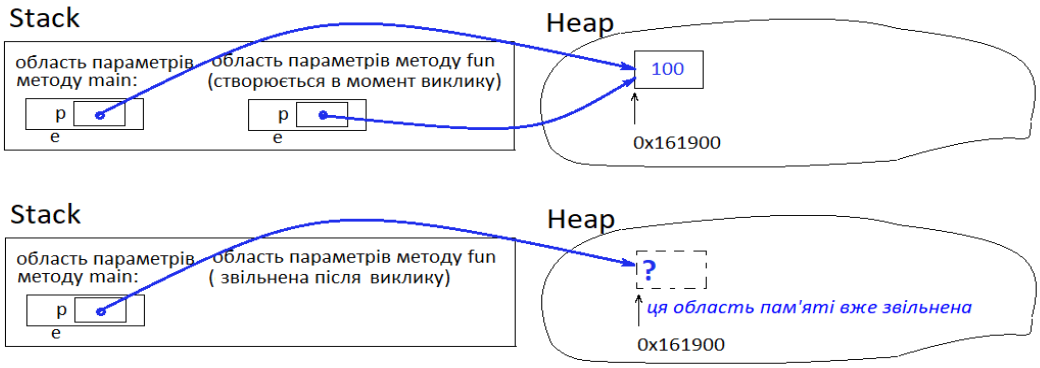
1.
```
    // Клас  Example містить вказівник та демонструє
    // необхідність конструктора копіювання:
#include <iostream>
using namespace std;

class Example       //  Це заголовок класу
{
    public:
     int * p;       // полем класу є вказівник
     Example (int val = 0)     // конструктор, створює та ініціалізує
                    // динамічну змінну, на яку вказує вказівник p;
     {
          p = new int;          // створили динамічну змінну
          *p = val;             // присвоїли їй значення
          // вивели вказівник та значення змінної
          cout << "Create Example: " << p << " " << *p << endl;
     }
     ~Example ()                // деструктор, він знищує динамічну
                                // змінну - звільняє пам'ять
     {
          cout << "Delete Example " << p << " " << *p << endl;
          delete p;
     }
};
void fun (Example e) // функція приймає екземпляр класу за значенням
{
```

```
      // виводимо вказівник та значення змінної одержаного екземпляру
      cout << "In fun: " << e.p << " " << *(e.p) << endl;
}
int main()
{
      Example e (100);     // створили екземпляр класу
      fun (e);             // передали його у функцію
      // вивели вказівник та значення змінної
      cout << "After fun: " << *(e.p) << endl;
}
```

Прослідкуємо за результатом виконання – на екрані послідовно з'являться такі повідомлення (безумовно, реальні адреси у вас будуть інші):

```
Create Example: 0x161900 100
In fun: 0x161900 100
Delete Example 0x161900 100
After fun: 0x161900 1447184
Delete Example 0x161900 1447184
```

Перше повідомлення надійшло від конструктора, який, створюючи екземпляр **e**, виділив пам'ять під динамічне поле **p** та присвоїв йому передане значення **100**. Це значення при одному конкретному виконанні на моєму комп'ютері знаходиться в області **Heap** за адресою **0x1f1900**. Наступне повідомлення надходить від функції **fun** – зверніть увагу, у створеній в стеку локальній копії екземпляру **e** поле-вказівник **p** має ту саму адресу! Адже системний конструктор копіювання при створенні копії екземпляру просто її скопіював. І саме за цією адресою пам'ять буде звільнена деструктором після завершення роботи функції **fun**. Що ми бачимо в повідомленні, яке виводиться у функції **main After fun:** ? Вказівник із значенням **0x161900** тепер вказує на недоступну, вже звільнену деструктором область пам'яті, тому значення, яке знаходиться по цій адресі – це просто «сміття»:



Для подібних класів, які містять вказівники, необхідно створити конструктор копіювання, який буде не копіювати вказівник, а виділить нову область пам'яті і вже в неї запише копію значення об'єкту, на який вказує цей вказівник. Додамо в наш клас **Example** конструктор копіювання. Для визначення конструктора копіювання необхідно дотримуватись особливого синтаксису:

```
<ім'я_класу> (const <ім'я_класу> & );
```

Параметром конструктора копіювання є ***стала*** змінна-посилання на екземпляр класу. Його призначення – коректне створення копії екземпляру. Ще раз наголосимо, що особливо важливо це у випадку, коли членами класу є вказівники, пам'ять під які виділяється

оператором **new**. Адже тоді поелементне копіювання копіює вказівник (поверхневе копіювання), а не об'єкт, на який він посилається, – для цього необхідне глибоке копіювання.

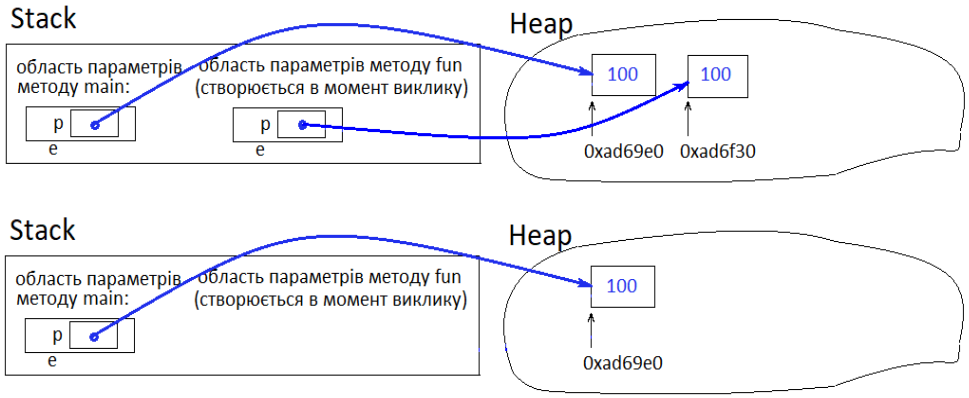
Включимо конструктор копіювання в класу **Example**, він має виглядати так:

```
Example (const Example & e)    // конструктор копіювання –
                               // створює нову динамічну змінну, копію існуючої
{
      p = new int;     // виділяємо нову область пам'яті
      *p = *(e.p);     // записуємо в нього значення, на яке вказує
                       // вказівник переданого екземпляру
      // друкуємо новий вказівник для копії значення
      cout << "Create Copy for Example: " << p << " " << *p << endl;
}
```

Тепер результат виконання зовсім інший:

```
Create Example: 0xad69e0 100
Create Copy for Example: 0xad6f30 100
In fun: 0xad6f30 100
Delete Example 0xad6f30 100
After fun: 0xad69e0 100
Delete Example 0xad69e0 100
```

Бачимо, що копія екземпляру, створена у стеку при виклику функції **fun**, тепер зберігає змінну, по іншій адресі. Вона буде звільнена після завершення функції, а область пам'яті, де зберігалось значення екземпляра **e** у функції **main** залишилась неушкодженою.



Залишилось додати, що конструктор копіювання також буде викликаний, якщо створити новий екземпляр, як копію існуючого, наприклад, таким чином: **Example ee (e);**.

6. Приклад проекту, в якому створений клас, що реалізує одновимірний масив із перевантаженням операцій над масивами.

Розглянемо клас **Array**, який реалізує масив (для простоти із цілими елементами). Він міститиме два закритих поля – кількість елементів в масиві та вказівник на початок цього масиву. Масиви з однаковою кількістю елементів можна очевидним чином додавати, тож перевантажимо в класі бінарну операцію додавання. Оскільки так само очевидно, що можна помножити число на масив – перевантажимо бінарну операцію множення дійсного числа на масив. Перевантажимо також унарний мінус – операцію переходу до масиву з протилежними елементами, операцію постфіксного інкременту, екзотику якого було обговорено, і ще операцію присвоєння, оскільки для нашого класу це необхідно. Включимо в клас допоміжні

функції – присвоєння елементам масиву початкових значень (нехай це будуть випадкові числа з діапазону [0..20]) та виведення значень елементів масиву на екран. У проект включимо три файли (алгоритм створення проекту був описаний у попередній лекції): **Array.h** – файл із визначенням класу, **Array.cpp** – файл із визначенням методів класу та **main_Array.cpp** – файл із функцією **main**, в якій ми тестуємо методи класу. Всі необхідні коментарі є просто в коді.

Приклади

2. `// Проект класу Array містить такі файли:`

Файл **Array.h:**

```
#ifndef ARRAY_H
#define ARRAY_H
class Array   // цей клас реалізує динамічний масив
{
   private :
     int size;  // кількість елементів  в масиві
     int *p;    // вказівник на початок масиву
   public :
     Array (int n = 1);  // декларація конструктора
     Array (const Array&);  // декларація конструктора копіювання
     ~Array ();       // декларація деструктора
     void Init ();   // задає елементи масиву - випадкові значення
     void Print ();  // виводить на екран масив поелементно
     Array operator+ (const Array &op2);  // додавання масивів
     Array operator= (const Array &op2); // присвоєння масивів
// декларація дружньої функції для множення числа на масив
     friend Array operator* (double op1, const Array &op2);
     Array operator- ();    // унарний мінус
     Array operator++ (int fict);   // постфіксний інкремент
     int operator[] (int index);   // індексатор - повертає елемент
     // масиву із вказаним індексом
};
#endif
```

Файл **Array.cpp:**

```
#include "Array.h"
#include <iostream>
#include <cstdlib>  // для функцій srand та rand
#include <ctime>    // для функції time
using namespace std;
//////////////////////////////////////////////////////////
Array :: Array (int n)   // реалізація конструктора
{
     if (n > 0) size = n;  // не пропускаємо невірні вхідні дані
     else n = 1;
     size = n;
     p = new int [size];  // створили масив
//   cout << "Create Array with " << size << " elements" << endl;
}
```

```
//////////////////////////////////////////////////////////
Array :: ~Array ()          // реалізація деструктора
{
     delete [] p;
//   cout << "Destruct Array with " << size << " elements" << endl;
}
//////////////////////////////////////////////////////////
Array :: Array (const Array& A)  // конструктор копіювання
{
     size = A.size;
     p = new int [size];    // створили масив для копії елементів
     for (int i = 0; i < size; i++)
        p[i] = A.p[i];   // скопіювали елементи в нову область пам'яті
//   cout << "!!! Create copy for Array with " << size << " elements"
<< endl;
}
//////////////////////////////////////////////////////////
void Array :: Init ()   // метод ініціалізації випадковими значеннями
{
     srand (time(0));     // встановлюємо базу для датчика
                          // псевдовипадкових чисел
     for (int i = 0; i < size; i++)
        p [i] = rand () % 21;  // випадкові числа в діапазоні [0..20]
}
//////////////////////////////////////////////////////////
void Array :: Print ()    // реалізація виводу масиву на екран
{
     for (int i = 0; i < size; i++)
        cout << p[i] << "\t"; // виводимо елементи через табулятор
     cout << endl;
}
//////////////////////////////////////////////////////////
Array Array :: operator+ (const Array &op2)
{
     if (op2.size == size)     // якщо масиви мають однакову
                    // кількість елементів, їх можна додавати
     {
          Array temp(*this);  // створили робочий масив
                              // як копію поточного екземпляру
          for (int i = 0; i < size; i++)
               temp.p[i] += op2.p[i];     // додали два масиви
          return temp;
     }
     else           // якщо кількість елементів в масивах різна -
                    // додавання неможливе
     {
          cout << "Illegal operation" << endl;
          return *this; // повертаємо поточний екземпляр як результат
     }
```

```
}
//////////////////////////////////////////////////////
Array Array :: operator= (const Array &op2)
{
     if (this == &op2) return *this; // контроль самоприсвоєння
     else if (size == op2.size) // якщо кількості елементів однакові
          for (int i = 0; i < size; i++)
               p[i] = op2.p[i];     // просто переписуємо елементи
                                    // в поточний масив
     else // якщо кількості елементів різні - присвоєння неможливе
          cout << "Illegal operation!" << endl;
     return *this;                  // повертаємо поточний масив
}
////////  це реалізація функції, продекларованої в класі, як дружня
Array operator* (double op1, const Array &op2)
{
     Array temp(op2);    // створили робочу змінну - копію другого
                         // операнду (тобто масиву) op2
     for (int i = 0; i < op2.size; i++)
        temp.p[i] *= op1;     // кожний елемент масиву помножили
                              // на перший операнд - число
     return temp;
}
//////////////////////////////////////////////////////
Array Array :: operator- ()
{
     return -1*(*this);       // використовуємо вже перевантажену
                              // операцію множення числа на масив
}
//////////////////////////////////////////////////////
Array Array :: operator++ (int fict)
{
     Array temp (*this);      // зберігаємо поточний масив
     for (int i = 0; i < size; i++)
        ++p[i];     // елементи поточного масиву інкрементуємо
     return temp;   // але повертаємо попередні значення
}
//////////////////////////////////////////////////////
int Array :: operator[] (int index)
{
     if ((index >= 0) && (index < size))     // якщо індекс
                              // знаходиться у допустимих межах
        return p[index];      // повертаємо відповідний елемент
     else                     // інакше - повідомляємо про помилку
     {
          cout << "Index is out of range" << endl;
          return 0;
     }
}
```

```
//////////////////////////////////////////////
```

І нарешті файл **main_Array.cpp:**

```
#include <iostream>
#include "Array.h"   //  підключили декларацію класу
using namespace std;
int main ()
{
     Array A (4), B (4); // створили 2 масиви по 4 елементи
     A.Init();           // задали елементи першого масиву
     system ("PAUSE");  // системна затримка - щоб згенерувались
                        // різні випадкові значення для двох масивів
     B.Init();
     A.Print();
     cout << "1)==========================\n";
     B.Print();
     Array C;   // створили масиви по умовчанню з 1 елементом
     C = A + B; // присвоїли йому результат додавання масивів
     cout << "2)==========================\n";
     C.Print(); // вивели на екран суму масивів
     cout << "3)==========================\n";
     C = 2*A;   // присвоїли результат множення 2 на масив A
     A.Print(); // вивели на екран масив A
     C.Print(); // вивели на екран результат множення
     cout << "4)==========================\n";
     C = -B;    // присвоїли результат -B
     B.Print(); // вивели на екран масив B – він не змінився
     C.Print(); // вивели на екран масив C, протилежний до B
     cout << "5)==========================\n";
     C = B++;   // постфіксний інкремент
     B.Print(); // вивели на екран масив B – його елементи збільшені
     C.Print(); // вивели на екран масив C – він збігається з
                // попереднім значенням иасиву B
     cout << "6)==========================\n";
     cout << C[1] << "  " << C[3] << endl;  // індексатор повертає
                                            // відповідні елементи
     return 0;
}
```

Ось результат одного із запусків цієї програми (елементи масивів – випадкові, але ми перевіряємо правильність виконання операцій з масивами):

```
Для продолжения нажмите любую клавишу . . .
3       8       17      19
1)==========================
10      14      16      5
2)==========================
13      22      33      24
3)==========================
3       8       17      19
6       16      34      38
4)==========================
```

```
10      14      16      5
-10     -14     -16     -5
5)==========================
11      15      17      6
10      14      16      5
6)==========================
14  5
```

Контрольні питання.

1. Що таке – перевантаження операцій в класі?
2. Як перевантажуються бінарні операції?
3. Як перевантажуються унарні операції?
4. Як перевантажуються префіксний та постфіксний інкременти та декременти?
5. Що таке конструктор копіювання, коли він потрібний?
6. Чи можна перевантажити операцію множення числа на масив в класі **Array** перевантажити операторною функцією-членом класу? Чому?

Завдання для самопідготовки.

1. Перенесіть вміст вказаних файлів у проект та відпрацюйте його.
2. Перевантажте в класі **Array** операцію віднімання масивів.
3. Перевантажте в класі **Array** операцію префіксного інкременту.
4. Перевантажте в класі **Array** операцію множення масиву на число. Які є способи це зробити?

Лекція 12. Статичні члени класу. Похідні класи, спадкування.

1. Статичні члени класу.

Познайомимось з особливими членами класу – так званими ***статичними членами***. Вони позначаються в класі службовим словом **static**. Особливість їх полягає в тому, що вони не належать жодному екземпляру класу, а використовуються всіма ними спільно.

Статичні дані-члени (поля) класів використовуються для збереження даних, спільних для всіх екземплярів класу, наприклад, кількість екземплярів класу, вказівник на вершину динамічного списку, деяку глобальну для всього класу константу, тощо. Декларація статичного поля класу має певні нюанси. По-перше, замало просто продекларувати статичне поле в класі, необхідно ще виділити для нього пам'ять, адже це поле не належить екземплярам. Для цього декларація статичного поля повторюються поза тілом класу, в області визначення глобальних змінних. При цьому можна додати ініціалізацію статичного поля, інакше за умовчанням, воно матиме нульове значення. По-друге, службове слово **static** поза класом не повторюється. Звертатись до статичного поля можна як через ідентифікатор будь-якого екземпляра, так і через тег класу і оператор області видимості **::** .

Приклади

1.
```
// Клас Example, що містить статичний член класу – поле num
#include <iostream>
using namespace std;
class Example
{
   public :
```

```
    static int num;  // декларація статичного поля класу
    int key;
    Example (int key_) : key (key_) {} // конструктор –
        // використовуємо список ініціалізації: полю key
        // присвоюється передане як параметр значення key_
};
//////////////////////////////////////////////////////////////
int Example :: num = 1; // виділення пам'яті під статичне поле
// поза межами класу (в разі відсутності ініціалізації поле = 0)
//////////////////////////////////////////////////////////////
int main(int argc, char *argv[])
{
  Example e (1), f (10);           // створили два екземпляри класу
  cout << Example :: num << endl; // вивели значення статичного поля
                                  // через ім'я класу
  Example :: num = 100;   // присвоїли значення статичному полю
  cout << e.num << endl; // вивели значення статичного поля
                         // через ім'я екземпляру
  cout << f.num << en
  e.num = 1000;          // присвоїли значення через ім'я екземпляру
  cout << e.num << endl;
  cout << f.num << endl;
  return 0;
}
```

Зауваження. Виділення пам'яті для статичного поля необхідне – навіть, якщо саме поле продеклароване в класі як закрите (**private**), пам'ять для нього все рівно виділяється поза класом, можливо, навіть разом з ініціалізацією! Це виглядає досить суперечливо, проте це так. Крім того, прийміть для себе, як більш наочний, варіант звертання до статичних членів через ім'я класу.

Результат виконання цього прикладу такий:

```
1
100
100
1000
1000
```

Статичні функції-члени класів використовуються тільки для звертання до статичних даних-членів і не можуть використовувати звичайні дані та методи класу, адже вони не прив'язані до екземпляру, отже, їм не не може бути переданий вказівник **this**.

Звертання до статичних функцій так само може відбуватись і через ім'я класу, і через ідентифікатор екземпляру.

Слід зауважити, що звичайні функції-члени класу мають право працювати із статичними членами класу. Конструктор та деструктор в C++ статичними бути не можуть.

Змінемо визначення класу з попереднього прикладу, додавши в нього статичну функцію, яка при кожному звертанні буде збільшувати статичне поле на одиницю.

Приклади

2.
```
    // Клас Example, що містить статичні члени класу – поле num
    // та функцію increase, яка збільшує це статичне поле
```

```
#include <iostream>
using namespace std;
class Example
{
   public :
     static int num;  // декларація статичного поля класу
     static void increase () {num++; }  // визначення статичної
                                        // функції
     int key;
     Example (int key_) : key (key_) {} // конструктор
};
//////////////////////////////////////////////////////////////
int Example :: num = 1; // виділення пам'яті під статичне поле
//////////////////////////////////////////////////////////////
int main(int argc, char *argv[])
{
  Example e (10);
  e.increase();     // виклик статичної функції через екземпляр
  cout << Example :: num << endl;
  Example :: increase ();// виклик статичної функції через ім'я класу
  cout << e.num << endl;
  return 0;
}
```

В результаті виконання цього прикладу одержимо такий вивід:

```
2
3
```

Приклади

3.

```
// Клас Counter для підрахунку існуючих екземплярів цього класу
#include <iostream>
using namespace std;
class Counter  // клас рахує наявні екземпляри
{
   public :
    static int count;     // декларація статичного поля - лічильник
    int key;              // номер екземпляру
    Counter (int key_) : key (key_) // конструктор
     {
          count++;  // збільшує лічильник при створенні екземпляру
     // та виводить його номер та загальну кількість екземплярів
          cout << "Creare example key = " << key << " total: " <<
count << endl;
     }
     ~Counter ()  // деструктор
     {
          count--; // зменшує лічильник при знищенні екземпляру
     // та виводить його номер та загальну кількість екземплярів
     cout << "Delete example key = " << key << " total: " << count
<< endl;
```

```
    }
};
int Counter :: count; // виділення пам'яті під статичне поле

int main ()
{
  for (int i = 1; i <= 5; i++)
  {
     Counter* p = new Counter (i); // створюємо 5 динамічних
// екземплярів - вони будуть пораховані завдяки статичному полю
// count та його змінам в конструкторі та деструкторі
  }
     Counter c (100);    // створюємо ще один екземпляр
     return 0;
}
```

Зауваження. В циклі **for** буде створено 5 динамічних екземплярів класу (на жаль, вказівники на них будуть втрачені після циклу, і ми не зможемо коректно звільнити пам'ять, проте вважатимемо це повчальним результатом). При цьому конструктор щоразу виводитиме нам номер створеного екземпляру (він збігається із значенням змінної циклу) та загальну кількість екземплярів. Ще один, нединамічний, екземпляр буде створений після циклу і знищений деструктором в момент завершення функції **main**. Ось що буде виведено на екран в результаті виконання:

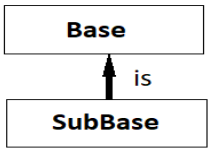
```
Creare example key = 1 total: 1
Creare example key = 2 total: 2
Creare example key = 3 total: 3
Creare example key = 4 total: 4
Creare example key = 5 total: 5
Creare example key = 100 total: 6
Delete example key = 100 total: 5
```

2. Спадкування. Похідні класи.

Наразі ми розібрали можливості, які надає ідея інкапсуляції інформації в класах, коли інформацію зберігають закриті дані-члени (поля) класу, а їх опрацювання – це прерогатива відкритих функцій-членів класу, які утворюють його ***інтерфейс***. Завдяки такому підходу контролюється цілісність інформації в екземплярах класу. Були розглянуті численні приклади створення та роботи із класами.

Наступним кроком опанування засад об'єктно-орієнтованого програмування буде створення класів на базі вже існуючих. Це дозволяє при зміні постановки задачі не програмувати класи наново, а лише дещо вдосконалити або розширити їх можливості. Ця ідея носить назву спадкування класів, створення так званих ***похідних класів*** від вже існуючих, які розглядаються як ***базові*** при спадкуванні.

При цьому похідний (інколи – дочірній) та базовий (інколи батьківський) класи пов'язані співвідношенням «**є**» («**is-a**») : якщо **SubBase** – похідний клас від класу **Base,** то **SubBase** «**є**» **Base**. В похідний клас таким чином автоматично переносяться всі члени базового класу, проте доступ до них визначається видом спадкування.



Слід зазначити, що ця прозора ідея в мові C++ втілена з дуже потужними можливостями, не всі з яких були перенесені в наступні C-подібні мови. Тому і тут буде розібраний далеко не повний перелік можливостей спадкування в C++.

Перш ніж продовжити обговорення, познайомимось ще з одним службовим словом, яке визначає рівень доступу до членів класу (і один з видів спадкування також). Раніше нам було відомо, що члени класу можуть бути відкритими (**public**) або закритими (**private**). Ще один рівень доступу **protected** визначає захищені члени класу – тобто закриті для всіх частин програми, окрім самого класу та його нащадків.

Види спадкування (не використовуються в мові C#) та доступ до відповідних членів базового класу у похідному класі зазначені у таблиці:

Вид спадкування	Доступ до члена класу у базовому класі	Доступ до члена класу у похідному класі
public	**private** **protected** **public**	не доступний **protected** **public**
protected	**private** **protected** **public**	не доступний **protected** **protected**
private (діє за умовчанням)	**private** **protected** **public**	не доступний **private** **private**

Синтаксис визначення похідного класу:

```
class <тег похідного класу> : <вид_спадкування> <тег базового класу>
{
        // тіло похідного класу
};
```

Зафіксуємо деякі правила спадкування

1. У похідному класі доступні всі відкриті (**public**) та захищені (**protected**) члени базового класу. Закриті (**private**) члени базового класу недоступні у похідному.
2. Конструктори та деструктор ***не спадкуються*** похідним класом.
3. При створенні екземпляру похідного класу ***автоматично викликається конструктор базового класу*** і лише ***потім конструктор похідного***. При знищенні екземпляру деструктори викликаються у ***зворотному*** порядку.
4. Якщо у похідному класі відсутній безпосередній виклик конструктора базового класу, то відбувається виклик конструктора за умовчанням (без параметрів) базового класу. В разі його відсутності виникає помилка.
5. Не спадкується також операція присвоєння, якщо вона була перевантажена у базовому класі.

Розглянемо простий ілюстративний приклад створення базового та похідного класу – спадкування відкрите, всі члени базового та похідного класів також відкриті. Конструктори та деструктори суто «декоративні», лише інформують про створення та знищення екзепляру, але саме це дозволить нам прослідкувати за порядком їх викликів.

Приклади

3.
```
// Просте однорівневе спадкування – клас SubBase походить від Base
#include <iostream>
using namespace std;
class Base                  // базовий клас Base
{
   public :
    int field               // деяке поле базового класу
    Base ()                 // конструктор базового класу
     {
          cout << "Create Base example" << endl;
     }
     ~Base ()               // деструктор базового класу
     {
          cout << "Delete Base example" << endl;
     }
};
class SubBase : public Base   // Похідний клас – походить від Base
{
   public :
    int additionalfield; // власне поле похідного класу
    SubBase ()           // конструктор похідного класу
     {
          cout << "Create SubBase example" << endl;
     }
     ~SubBase ()         // деструктор похідного класу
     {
          cout << "Delete SubBase example" << endl;
     }
};
int main()
{
  SubBase sb;        // створений екземпляр похідного класу
// він містить і власне поле, і успадковане поле базового класу
  cout << sb.field << " " << sb.additionalfield << endl;
  return 0;
}
```

Результат виконання цього прикладу такий:

```
Create Base example
Create SubBase example
0 0
Delete SubBase example
Delete Base example
```

Ми бачимо, що для створення екземпляру **sb** похідного класу **SubBase** був викликаний спочатку конструктор базового класу, і лише потім – конструктор похіного класу (тобто при

створенні цього екземпляру спочатку була створена умовний «фундамент» конструктором **Base**, а потім він був «добудований» конструктором **SubBase**). Деструктори спрацьовують рівно навпаки – спочатку «руйнується надбудова», а потім – основа. Цілком логічна схема, чи не так? Зверніть також увагу, що екземпляр **sb** містить два поля – і успадковане поле **field** базового класу, і власне поле **additionalfield**. Їх значення нульові, адже вони не були проініціалізовані.

Спробуємо ускладнити цей приклад, ввівши в базовий клас додаткові поля із рівнями доступу **private** та **protected**. Це призведе до необхідності включити параметри в конструктор базового класу для їх ініціалізації. І тоді виникне проблема, зазначена у 4 пункті правил спадкування, – автоматично система спроможна викликати лише конструктор базового класу без параметрів. Щоб подолати її є дві можливості – передбачити умовчання для кожного параметра конструктора базового класу, або забезпечити безпосередній виклик конструктора базового класу із необхідним списком параметрів. Оскільки з першою можливістю все зрозуміло, обговоримо застосування другої.

Для використання можливості явного виклику конструктору базового класу **Base** у похідному класі **SubBase**, використовується така синтаксична конструкція:

```
SubBase (<параметри конструктора>): Base(<параметри конструктора>)
{
     // тіло конструктора похідного класу
}
```

Реалізуємо в прикладі вказані зміни. Включимо також в похідний клас функцію-член, яка використовуватиме захищений член базового класу та (для перевірки прав доступу) спробує звернутись до закритого члену базового класу.

Приклади

4.
```
     // Просте однорівневе спадкування – виклик конструктора базового
     // класу та контроль доступу до членів базового класу
#include <iostream>
using namespace std;
class Base
{
   public :
    int field_1;          // відкрите поле базового класу
   protected :
    double field_2;       // захищене поле базового класу
   private :
    string field_3;       // закрите поле базового класу
   public :
    Base (int f1, double f2, string f3) : // конструктор базового
                // класу тепер має параметри
field_1 (f1),  field_2 (f2), field_3 (f3) // ініціалізація списком
     {
          cout << "Create Base example" << endl;
     }
     ~Base ()  // деструктор базового класу
     {
          cout << "Delete Base example" << endl;
     }
};
```

```
class SubBase : public Base    // Похідний клас – походить від Base
{
   public :
    int additionalfield;       // відкрите поле базового класу
// конструктор похідного класу
    SubBase (int f1, double f2, string f3, int addf) :
     Base (f1, f2, f3)    // явний виклик конструктора базового класу
     {
          additionalfield = addf;
          cout << "Create SubBase example" << endl;
     }
     ~SubBase ()  // деструктор похідного класу
     {
          cout << "Delete SubBase example" << endl;
     }
     void fun ( )
     {
          double field_2 = -field_2;     // допустиме звертання до
                                         // захищеного члена базового класу
//        cout << field_3 << endl;       // недопустиме звертання до
                                         // закритого члена базового класу
     }
};
int main( )
{
  SubBase sb (1, 2.5, "test", 100); // екземпляр похідного класу
  cout << sb.field_1 << " " << sb.additionalfield << endl;
  return 0;
}
```

Зауваження. В конструкторі базового класу використана вже згадувана ініціалізація списком трьох полів цього класу. Конструктор похідного класу має 4 параметри, 3 з них необхідні для ініціалізації полів базового класу (реально поле **field_3** буде недоступне у похідному класі), ще один параметр буде використаний для ініціалізації власного поля похідного класу. Ще раз перегляньте, як здійснюється явний виклик конструктора базового класу. Зверніть також увагу на функцію **fun** – вона введена в цей приклад, щоб продемонструвати доступність в похідному класі поля **field_2**, яке мало модифікатор доступу **protected** в базовому класі, і недопустимість звертання до приватного поля **field_3** базового класу (це звертання закрите коментарем). Ось результат виконання цього прикладу:

```
Create Base example
Create SubBase example
1 100
Delete SubBase example
Delete Base example
```

Ще раз звернемо увагу на список ініціалізації – в багатьох випадках він є лише альтернативою безпосередньому присвоєнню початкових значень полям в конструкторі. Синтаксично список ініціалізації – це просто перелік полів класу із початковими значеннями, що вказуються в дужках безпосередньо поруч із ними. Елементи списку відокремлюється від

списку параметрів конструктора розділовою двокрапкою, а один від одного – комами. Проте інколи виникають ситуації, коли безпосереднє присвоєння значень полям неможливе, і тоді список ініціалізації є необхідністю. Наприклад, якщо членом класу є **reference**-змінна, або поле визначене як константне, або поле є екземпляром деякого іншого класу.

Контрольні питання.

1. Що таке статичний член класу, для чого і як він використовується?
2. Як визначити статичне поле в класі?
3. Чи може звичайна функція-член класу використовувати статичні функції та поля класу?
4. Чи може статична функція-член класу використовувати нестатичні функції та поля класу?
5. Що означає термін «спадкування»?
6. Що таке базовий та похідний класи?
7. Яким є порядок виклику конструкторів та деструкторів при створенні екземпляру похідного класу?
8. Який саме конструктор базового класу викликається при створенні екземпляру похідного класу?
9. Які члени класу базового класу доступні у похідному? Як це залежить від виду спадкування?

Завдання для самопідготовки.

1. Виконайте приклад 4, виключивши параметр **f3** із списку параметрів конструктора для ініціалізації недоступного поля **field_3**. Якими способами це можна зробити?
2. Створіть власний приклад, в якому фігурують базовий та похідний класи.

Лекція 13. Поліморфізм та віртуальні функції.

1. Зв'язки між базовим та похідним класом.

Очевидно, що базовий та похідний клас пов'язані вже тим, що похідний клас має в своєму розпорядженні всі відкриті та захищені члени базового класу. Другий тип «родинного» зв'язку (пам'ятаємо про співвідношення «**Є**») між цими класами полягає в тому що:

- екземпляр базового класу можна створити як екземпляр похідного;
- посилання на базовий клас може посилатись на похідний;
- вказівник на базовий клас може вказувати на похідний.

Покажемо, що це означає на прикладі (проте треба розуміти, що у всіх цих випадках базовий клас не має доступу до членів похідного класу):

Приклади

1.
```
// Встановлюємо зв'язки між базовим та похідним класами
#include <iostream>
using namespace std;
class Base
{
   public :
    int field;
    void test ()          // функція базового класу
    {
        cout << "Test Base\n";
    }
```

```
};
class SubBase : public Base
{
   public :
    int additionalfield;
};
int main( )
{
  SubBase sb;         // створений екземпляр похідного класу
  Base b = sb;        // екземпляр базового класу створений
                      // через присвоєння екземпляру похідного
  Base& rb = sb;      // посилання на екземпляр базового класу є
                      // посиланням на екземпляр похідного
  Base *pb = &sb;     // вказівник на екземпляр базового класу є
                      // вказівником на екземпляр похідного
  pb = new SubBase ();    // екземпляр базового класу створений як
                          // екземпляр похідного
  // присвоєння у зворотний бік можливе лише явним приведенням типу:
  SubBase *psb = (SubBase *) pb;  // вказівник на Base приведений
                          // до типу вказівника на SubBase
  // sb = b;              // таке присвоєння неможливе!
  return 0;
}
```

Цілком зрозуміла і заборона присвоєння у зворотний бік, адже, якщо б присвоєння **`sb = b;`** було б можливе, екземпляр похідного класу **`sb`** не міг би містити власного поля **`additionalfield`**!

Та обставина, що посилання та вказівники базового класу можуть вказувати на екземпляри похідних класів, приводить до низки цікавих можливостей – зокрема методи, які мають параметрами посилання або вказівник ***на базовий клас***, можуть викликатись із аргументами-екземплярами ***похідних класів***. Якщо додати в попередній приклад визначення функції

```
void fun (Base &b)// параметр методу – посилання на базовий клас
{
     b.test ();
}
```

то в **`main`** допустимі виклики **`fun (b); fun (*pb); fun (rb);`** або **`fun (sb);`** Проте в будь-якому разі функція **`fun`** викликатиме метод **`test`** базового класу.

З іншого боку метою створення похідних класів є необхідність покращення або зміни контенту базового класу. Це означає, що може виникати необхідність, щоб функції, визначені в базовому класі, виконували в похідному класі дещо інше або додаткове. З метою подібного експерименту замість функції **`test`** базового класу, яка спадкується у похідний клас, створимо у класі **`SubBase`** функцію з тим самим іменем **`test`**, проте з іншим «змістом»:

```
void test ()          // це буде функція похідного класу
{
     cout << "Test SubBase\n";
}
```

Ідентична по сигнатурі функція **test** в похідному класі «перекриє» відповідну функцію базового класу при звертаннях до екземплярів похідного класу (подібно до того, як локальна одноіменна змінна «перекриває» глобальну).

У функцію **main** помістимо такі виклики:

```
int main( )
{
  SubBase sb;          // створений екземпляр похідного класу
  Base b = sb;         // екземпляр базового класу створений
                       // присвоєнням екземпляру похідного
  Base& rb = sb;       // посилання на екземпляр базового класу –
                       // посилання на екземпляр похідного
  Base *pb = &sb;      // вказівник на екземпляр базового класу –
                       // вказівник на екземпляр похідного
  pb = new SubBase ();   // екземпляр базового класу створений як
                         // екземпляр похідного
  // присвоєння у зворотний бік можливе лише явним приведенням типу:
  SubBase *psb = (SubBase *) pb;  // вказівник на Base приведений
                         // до типу вказівника на SubBase
  // sb = b;             // таке присвоєння неможливе
  fun (b);             // спрацює метод test базового класу
  fun (sb);            // спрацює метод test базового класу
  fun (*pb);           // спрацює метод test базового класу
  fun (rb);            // спрацює метод test базового класу
  sb.test();           // спрацює метод test похідного (!) класу
  return 0;
}
```

Результат виконання цього прикладу буде такий:

```
Test Base
Test Base
Test Base
Test Base
Test SubBase
```

Видно, що виклик методу **test** з похідного класу спрацьовує лише в останньому рядку, коли він викликається безпосередньо для екземпляру **sb** похідного класу. В усіх інших випадках, коли відбувається приведення до типу базового класу, має місце і прив'язка до методу **test** базового класу.

2. Поняття про раннє та пізнє зв'язування.

Спробуємо, хоч і дуже спрощено та схематично, але все ж таки зрозуміти, як реально відбувається «прив'язка» методу до його виклику. При компіляції всі функції перетворюються у двійкові коди і зберігаються в окремих місцях пам'яті. Адреси цих місць зберігання заносяться у спеціальні таблиці у відповідності до імен функцій. Коли в коді програми зустрічається виклик функції, компонувальник (редактор зв'язків) по імені фукції вибирає з таблиці належне місце в пам'яті, де зберігається потрібна функція та ініціює її виконання. Ця прив'язка адреси місця зберігання функції до її виклику називаається ***зв'язуванням*** виклику функції. Тут варто нагадати про технологію зміни імен функцій **name mangling**, яка використовується компіляторами C++ для перевантаження функцій з різною сигнатурою. У

згадану таблицю заносяться вже змінені відповідно до списків параметрів імена функцій. Таким чином виклики всіх звичайних функцій та функцій-членів класів обробляються ***перед*** підготовкою до виконання **exe**-файлу програми. Ця процедура носить назву ***раннього*** або ***статичного зв'язування***. Саме тому ми бачимо виклики функції **test** базового класу, адже у відповідності до типу параметру (**Base**) у функції **fun** раннє зв'язування забезпечує передачу управління саме функції базового класу.

Якщо ж зв'язування відкладається до ***часу виконання програми***, то воно носить назву ***пізнього*** або ***динамічного зв'язування*** (в мові C# відподна технологія носить назву ***динамічного поліморфізму***). Стає зрозумілим, що це саме те, що хотілося б реалізувати для класів-нащадків. Уявіть, що для розв'язання деякої задачі є базовий клас і ряд функцій, які працюють із екземплярами цього класу та його функціями-членами. Уявіть також, що в похідному класі був дещо розширений функціонал базового класу і є одноіменні функції-члени похідного класу, які мають більше можливостей порівняно із своїми «попередницями». Безумовно, потрібно, щоб функції, які мали параметрами посилання або вказівники на базовий клас, одержавши посилання або, відповідно, вказівник на похідний клас, викликали би тепер потрібні функції похідного класу. Це стає можливим, якщо використати службове слово **virtual** біля імені функції у базовому класі!

3. Віртуальні функції.

Пізнє зв'язування стосується функцій-членів (методів), які називаються ***віртуальними функціями***. Віртуальна функція (**virtual**) оголошується в базовому класі і може бути ***перевизначена*** у похідних класах. Сукупність класів, в яких визначається і перевизначається віртуальна функція, називається ***поліморфним кластером***. У межах цього кластеру об'єкт (екземпляр) пов'язується із конкретною віртуальною функцією-членом ***під час виконання програми***. Звичайна функція-член також може бути перевизначена у похідному класі, як це було зроблено у змінах до попереднього прикладу. Проте без атрибуту **virtual** до неї буде застосоване, як ми переконались, лише раннє зв'язування.

Термін «поліморфізм» означає втілення принципу – «один інтерфейс але різні реалізації», тобто однакові за сигнатурою функції із базового та похідного класу демонструють різну поведінку в залежності від типу екземпляру (а цей тип може бути визначений після компіляції, вже в процесі виконання програми), який здійснює звертання до них.

Внесемо ще одну-єдину зміну у попередній приклад – додамо атрибут **virtual** до функції **test** базового класу.

Приклади

2.
```
// Знайомимось з віртуальною функцією
#include <iostream>
using namespace std;
class Base
{
   public :
    int field;
    virtual void test ()       // віртуальна функція базового класу
    {
        cout << "Test Base\n";
    }
};
class SubBase : public Base
```

```
{
   public :
    int additionalfield;
    void test ()// в похідному класі функція базового класу заміщена:
    {
        cout << "Test SubBase\n";
    }
};
////////////////////////////////////////////////
void fun (Base &b)  // параметр функції – посилання на базовий клас
{
     b.test ();
}
////////////////////////////////////////////////
int main( )
{
  SubBase sb;        // створений екземпляр похідного класу
  Base b = sb;       // екземпляр базового класу створений
                     // присвоєнням екземпляру похідного
  Base& rb = sb;     // посилання на екземпляр базового класу –
                     // посилання на екземпляр похідного
  Base *pb = &sb;    // вказівник на екземпляр базового класу –
                     // вказівник на екземпляр похідного
  pb = new SubBase ();   // екземпляр базового класу створений як
                         // екземпляр похідного
  // присвоєння у зворотний бік можливе лише явним приведенням типу:
  SubBase *psb = (SubBase *) pb;  // вказівник на Base приведений
                                  // до типу вказівника на SubBase
  // sb = b;                      // таке присвоєння неможливе
  fun (b);           // спрацює метод test базового класу
  fun (sb);          // спрацює заміщений метод test з похідного класу
  fun (*pb);         // спрацює заміщений метод test з похідного класу
  fun (rb);          // спрацює заміщений метод test з похідного класу
  sb.test();         // спрацює метод test похідного класу
  return 0;
}
```

Переконайтесь, що результат виконання тепер такий:

```
Test Base
Test SubBase
Test SubBase
Test SubBase
Test SubBase
```

Перший та останній виклики функції **fun** дають очевидні результати (ті самі, що й раніше). Решта викликів функції **fun** , яка приймає посилання на базовий клас, але в момент виконання одержує екземпляр, який був приведений до типу цього посилання від екземпляру похідного класу, приводить до виклику заміщеної функції **test**.

Слід зазначити, що використання пізнього зв'язування є достатньо складним механізмом, який вимагає суттєвих ресурсних витрат. Адже адреси зберігань в пам’яті віртуальних функцій знаходяться в окремих таблицях віртуальних функцій (ТВФ). Коли

створюється екземпялр класу, в якому є хоч одна віртуальна функція, то разом з таким екземпляром зберігається і посилання на ТВФ. І якщо для такого екземпляра викликається віртуальна функція, то місце її зберігання визначається із ТВФ. Це незначним чином, але дещо уповільнює виконання програми. Тому віртуальними слід робити лише такі функції, які ***дійсно є сенс перевизначати*** у похідних класах.

Окрім того, конструктори не можуть бути віртуальними – адже похідний клас не спадкує конструктор базового. А от деструктор може бути віртуальним. Користь віртуального деструктора розглянемо у наступному прикладі, висновком з якого може бути основне правило:

– ***Якщо клас містить хоч одну віртуальну функцію, деструктор класу теж слід зробити віртуальним.***

Отже, проаналізуємо наступний приклад.

Приклади

3.
```
// Чи вірно спрацьовують деструктори?
#include <iostream>
using namespace std;
class ScalarType    // базовий клас – скалярний тип
{
   public:
     virtual char type () // віртуальна функція – повертає символ,
                          // який означає тип
     {
          return 'U';    // символ означає деякий тип
     }
     ~ScalarType ()      // деструктор
     {
          cout << "Destruct example ScalarType " << endl;
     }
};
class FloatType : public ScalarType // похідний клас – дійсний тип
{
   public:
     char type ()        // заміщення віртуальної функції
     {
          return 'F';    // символ F означає дійсний тип
     }
     ~FloatType ()       // деструктор
     {
          cout << "Destruct example FloatType "<< endl;
     }
};
class DoubleType : public FloatType // похідний клас – довгий дійсний
{
     public:
     char type ()        // заміщення методу базового класу
     {
          return 'D';    // символ D означає довгий дійсний тип
     }
     ~DoubleType ()      // деструктор
```

```
    {
        cout << "Destruct example DoubleType "<< endl;
    }
};
int main ()
{
    ScalarType *p = new  FloatType ();
    delete p;
    cout << "1) ================="  << endl;
    p = new DoubleType ();
    delete p;
    cout << "2) ================="  << endl;
    DoubleType *pd = new DoubleType ();
    delete pd;
    cout << "3) ================="  << endl;
    return 0;
}
```

Зауваження. В цьому прикладі створена ієрархія з трьох класів – базовий клас визначає деякий скалярний тип, похідний клас першого рівня спадкування – дійсний тип і похідний від нього – довгий дійсний тип. В базовому класі є віртуальна функція, яка «сигналить» про тип свого об'єкта певним символом, в обох похідних класах ця функція перевизначена, оскільки має повертати символ у відповідності до свого типу. Класи використовують системні конструктори по умовчанню, а от деструктори в них визначені (вони просто повідомляють про знищення свого екземпляра). Як ми гарно пам'ятаємо, при створенні екземпляра похідного класу мають послідовно спрацювати конструктори базового класу та потім похідного. При знищенні екземпляру деструктори викликаються у зворотному порядку. Саме їх виклики ретельно проаналізуємо. В функції **main** створюється динамічний екземпляр **FloatType**, вказівник на який приводиться до типу вказівника на базовий клас. Цей динамічний екземпляр знищується, і можна очікувати повідомлення від деструктора **FloatType**, а потім – від деструктора **ScalarType**. Проте виводится лише повідомлення від деструктора базового класу! Далі створюється динамічний екземпляр **DoubleType**, при знищенні якого мали би спрацювати аж три деструктори, але знову спрацьовує лище деструктор базового класу. І лише третій динамічний екземпляр демонструє «правильну» послідовність викликів деструкторів. Ось результат виконання цього прикладу:

```
Destruct issue ScalarType
1) =================
Destruct issue ScalarType
2) =================
Destruct issue DoubleType
Destruct issue FloatType
Destruct issue ScalarType
3) =================
```

Такий стан справ є реальною проблемою, пов'язаною із неконтрольованими витоками пам'яті (**memory leaks**), адже не були коректно знищені ті частини екземплярів, які відповідають похідним класам. Причина в тому, що об'єкт знищується через вказівник на базовий клас, а базовому класу нічого невідомо про похідний – для деструкторів працює раннє зв'язування. Щоб позбутись цього ефекту, ***деструктор в базовому класі слід визначити як віртуальний***. Тоді і знищення екземпляру, адресованого вказівником, відбудеться коректно.

Додайте атрибут **virtual** перед деструктором у базовому класі – ви одержите зовсім інший результат, який демонструє, що тепер екземпляри знищуються коректно:

```
Destruct issue FloatType
Destruct issue ScalarType
1) ================
Destruct issue DoubleType
Destruct issue FloatType
Destruct issue ScalarType
2) ================
Destruct issue DoubleType
Destruct issue FloatType
Destruct issue ScalarType
3) ================
```

Взагалі кажучи, якщо клас передбачає спадкування, його деструктор завбачливо мав би бути визначений віртуальним. Проте йти на такі додаткові витрати варто саме тоді, коли в класі є принаймні одна віртуальна функція.

4. Абстрактний базовий клас.

Наразі нам відомі правила простого спадкування та більш складного поліморфного спадкування, яке включає використання віртуальних функцій. Наступний рівень складності – використання абстрактного базового класу. Необхідність в ньому виникає, коли потрібно описати об'єкти, які мають східну природу, проте важко визначити їх підпорядкування як базового та похідного класів. Наприклад, розглядаючи такі об'єкти, як прямокутник та ромб, неможливо встановити між ними відношення «**Є**» («**is-a**»), хоча й очевидно, що вони мають багато спільного: наприклад, поняття площі, повороту на площині. У таких випадках необхідно виділити у об'єктів все спільне і створити клас, який буде базовим для них всіх. Якщо реалізація окремих функцій можлива лише на рівні похідних класів, у базовому з допомогою спеціального синтаксису такі функції визначають як ***суто віртуальні функції*** **(pure virtual function)**. Така функція не має реалізації в базовому класі, отже, екземпляри такого базового класу неможливо створити. Сам клас при цьому називається ***абстрактним*** і призначений лише для створення похідних класів.

Розглянемо приклад, в якому абстрактний клас реалізує деяку просторову фігуру, що має об’єм. Проте визначити об’єм довільної фігури неможливо, тому відповідна функція буде продекларована як суто віртуальна. У похідних же класах, що реалізують конкретні фігури, ця функція обов’язково має бути «реалізована», тобто визначена.

Приклади

5.
```
// Знайомство з суто віртуальною фуекцією та абстрактрим класом
#include <iostream>
#include <cmath>
using namespace std;
class Figure_3D  // абстрактний клас із суто віртуальною функцією
{
   public:
   Figure_3D ()  // конструктор
   {
       cout << "Create Figure_3D " ;
   }
```

```
   virtual double volume () = 0;    // pure virtual function
};
class cube : public Figure_3D       // похідний клас - куб
{
   public:
   double side;                     // сторона куба
   cube (double s) : side (s)       // конструктор
   {
       cout << "cube" << endl;
   }
   double volume () // суто віртуальна функція обов'язково
                    // має бути реалізована в похідних класах!
   {
      return side*side*side;        // визначення об'єму куба
   }
};
class sphere : public Figure_3D     // похідний клас - сфера
{
   public:
   double radius;                   // радіус сфери
   sphere (double r) : radius (r)   // конструктор
   {
       cout << "sphere" << endl;
   }
   double volume () // суто віртуальна функція обов'язково
                    // має бути реалізована в похідних класах!

   {
      return 4*M_PI*radius*radius*radius/3;//визначення об'єму сфери
   }
};
int main ()
{
    //Figure_3D f;  // заборонено створювати екземпляр абстр. класу!
    cube c (10);    // створили куб із стороною 10
    sphere s (10);  // створили кулю із радіусом 10
    cout << "Volume of cube = " << c.volume () << endl;
    cout << "Volume of sphere = " << s.volume () << endl;
    //  проте можна створити екземпляр похідного класу
    //  та привести до типу абстрактного базового класу
    Figure_3D *p = new cube (1);
    // пізнє зв'язування працює
    cout << "Volume of cube = " << p->volume() << endl;
    return 0;
}
```

Ось результат виконання цього прикладу:

```
Create Figure_3D cube
Create Figure_3D sphere
Volume of cube = 1000
```

```
Volume of sphere = 4188.79
Create Figure_3D cube
Volume of cube = 1
```

Контрольні питання.

1. Що таке раннє та пізнє зв’язування?
2. Чому перевантаження функцій називається «простим поліморфізмом»?
3. Що таке віртуальна функція та віртуальний кластер?
4. Чи може конструктор бути віртуальним?
5. Коли виникає необхідність у віртуальному деструкторі?
6. Що таке суто віртуальна функція, яке її призначення?
7. Що таке абстрактний клас, як він використовується?

Завдання для самопідготовки.

1. Створіть клас **`Sportsman`** та похідний від нього клас **`Runner`**. В кожному класі має бути конструктор, що ініціалізує поля та повідомляє про створення екземпляру, та деструктор, що повідомляє про знищення екземпляру. Створіть в базовому класі віртуальний метод **`Training`** та перевизначте його у похідному класі. Уважно проаналізуйте його роботу.
2. Виконайте приклад 5, додавши ще один похідний клас **`pyramide`** – правильна трикутна піраміда. Клас має містити поля, що задають довжину сторони основи та висоту піраміди.

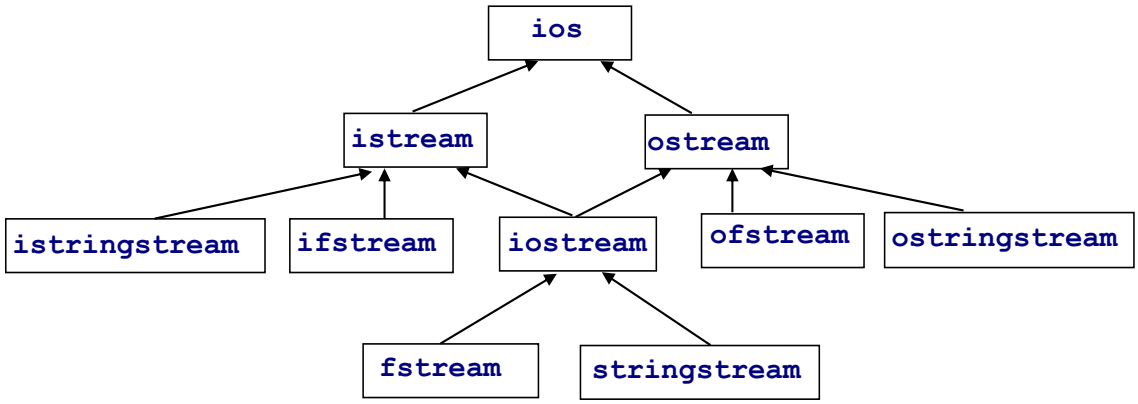
Лекція 14. Класи потоків введення та виведення в С++.

В лекціях 4 та 7 вже обговорювались деякі можливості потоків введення–виведення в С++. Зараз обговоримо це питання більш глибоко з «класовим» розумінням. Заголовочні файли **`iostream`** та **`fstream`** містять набори класів, що забезпечують так зване потокове введення та виведення інформації в стилі мови С++. З точки зору програми С++, введення та виведення являє собою потік байтів. При введенні програма вибирає байти з потоку вводу, при виведенні – відправляє байти в потік. Якщо програма орієнтована на роботу з текстом (або навіть з числовою інформацією в текстовому вигляді), кожний байт є представленням чергового символу. В загальному випадку байти потоку є двійковим представленням даних довільного типу. Байти потоку введення можуть поступати з клавіатури або з іншого зовнішнього пристрою пам'яті. Так само байти потоку виведення можуть бути відправлені на екран, принтер або ж зовнішній пристрій. Тобто потік діє як ***посередник*** між точкою відправлення інформації та точкою її обробки. Це дозволяє С++-програмі обробляти потоки незалежно від того, звідки інформація поступила або куди вона направляється. Образно потік можна собі уявляти як трубу, по якій спрямовуються байти інформації. Отже, для роботи з потоком його треба приєднати до програми та зв'язати з файлом, що містить або буде містити належну інформацію. Для більш ефективного введення та виведення інформації використовуються буфери, які відіграють роль резервуарів для накопичення інформації, щоб зменшити кількість звертань до зовнішніх пристроїв пам'яті та вирівняти швидкість операцій зчитування та запису.

1. Ієрархія класів потоків мови С++. Керування потоками.

На діаграмі нижче зображено гроно класів, що підтримує операції введення-виведення в С++. Як ми пам’ятаємо, напрям стрілки вказує напрямок дії співвідношення «Є», тобто відношення похідного класу до базового. Деякі заголовочні файли із цієї ієрархії вже

використовувались у прикладах попереднвх лекцій, так само, як і класи та їх методи із цих заголовочних файлів.



Базовий клас **ios** містить зокрема набір так званих ***прапорців***. Вони мають значення 1 (прапорець встановлений) або 0 (прапорець скинутий) і керують тим, в який спосіб потік обмінюється іноформацією з файлом. Є декілька категорій цих прапорців – для встановлення режиму роботи потоку (введення чи виведення), для визначення стану потоку (чи були помилки під час роботи потоку), для його форматування (тобто керування виглядом інформації, яка поставляється потоком). Більшість прапорців мають значення за умовчанням, змінити або встановити потрібні значення можна лише конструктором відповідного класу, при створенні потоку або методом **open**. Встановити значення декількох прапорців можна виразом, в якому їх імена об'єднані операцією **|** (бітове «АБО»). Для встановлення прапорця досить вказати його назву. Ось основні прапорці, що керують станом потоку:

ios::in – потік налаштовується на введення інфориації;
ios::out – потік налаштовується на виведення інфориації;
ios::trunc – попередній зміст файлу буде видалений при виведенні;
ios::app – попередній зміст файлу буде доповнений при виведенні;
ios::binary – потік налаштований на виведення у двійковому, а не текстовому форматі.

Наприклад, для створення потоку **fs** для запису інформації у текстовий файл **f1.txt** із доповненням його попереднього вмісту необхідно записати:

```
fstream   fs ("f1.txt", ios::out | ios::app);
```

або

```
ofstream   fs ("f1.txt", ios::app);
```

В останньому випадку екземпляр потоку **ofstream** вже налаштований на виведення за умовчанням, в обох випадках, також за умовчанням, потік вважається текстовим.

Ще один приклад створення потоку **fs** для читання із двійкового файлу **f2.bin**:

```
ifstream fs ("f2.bin", ios::in | ios::binary);
```

Потік може бути створений без прив'язки до файлу:

```
ofstream fs; // за умовчанням це буде потік для запису у файл
```

Сам файл при цьому може бути відкритий пізніше:

```
fs.open("f3.txt"); // Відкриваємо файл для запису з початку
```

При будь-якому способі відкриття файлу після завершення всіх операцій він має бути закритий: **`fs.close();`**

Для контролю за станом потоку використовується функції-члени:

- ✓ функція **`bad()`** повертає **`1`**, якщо при створенні екземпляру потоку **`fs`** виникла помилка:

```
if (fs.bad())
{
     cout << "Opening file error\n";
}
```

- ✓ функція **`clear()`** скидає прапорець помилки (інакше подальші спроби введення-виведення блокуються): **`fs.clear();`**
- ✓ функція **`is_open ()`** повертає **`0 (false)`**, якщо метод **`open ()`** не спрацював успішно:

```
if (fs.is_open ())
{
     // тут працюємо з файлом
     fs.close();     // на завершення – закриваємо файл
}
else
{
     cout << "Opening file error\n";
}
```

- ✓ функція **`good()`**, яка повертає **`1 (true)`** в разі успішного відкриття файлу:

```
if (fs. good())
{
     // тут працюємо з файлом
     fs.close();     // на завершення – закриваємо файл
}
else
{
     cout << "Opening file error\n";
}
```

- ✓ функція **`eof()`** повертає значення **`1 (true)`**, якщо під час введенння досягнутий кінець файлу – це означає, що всі подальші спроби введення із цього файлу будуть блоковані.

Потоки автоматично закриваються при завершенні програми. Однак при необхідності можна закрити потік функцією **`close()`**, а потім знову відкрити його, зв'язавши з іншим файлом, або і з тим самим, проте в іншому режимі.

Для форматування потоків також використовуються відповідні прапорці, функції-члени та ***маніпулятори***. Деякі маніпулятори з прикладами їх використання обговорювались в лекції 4. Нагадаємо тут лише деякі з них:

`setw ()`	Встановлення ширини поля виводу
`setprecision ()`	Кількість знаків після коми при виводі дійсних чисел

fixed	Встановлюється для виводу дійсних чисел у формате с фиксоваою крапкою та заданою кількістю розрядов праворуч від крапки (визначеним з допомогою функції-члена **precision ()** або маніпулятора **setprecision ()**
left	Вирівнювання вихідних даних по лівому краю поля
right	Вирівнювання вихідних даних по правому краю поля

2. Введення та виведення текстової інформації.

Одними з найпоширеніших операцій при роботі з файлами є введення та виведення текстової інформації або й числової інформації, яка представлена в текстовому форматі. Для введення та виведення довільних скалярних типів мови С++, а також об'єктів типу стрінг використовуються вже знайомі нам операції вибірки з потоку (**<<**) та вставки в потік (**>>**), а також функції-члени класу **ios** та похідних від нього класів. Наведемо тут деякі з цих функцій.

Якщо **ch** змінна типу **char**, то вона може бути прочитана із файлу, пов'язаного з потоком **fs**, однією із перевантажених функцій-членів **get ()**:

```
ch = fs.get ();
```

або

```
fs.get (ch);
```

Якщо **ch** масив розміру **size** типу **char**, то він може бути заповнений символами із файлу ще однією перевантаженою фукцією **get ()**:

```
fs.get (ch, size, '\n');
```

або

```
fs.getline (ch, size, '\n');
```

(останній параметр – не обов'язковий, він має умовчання)

Для запису у файл, пов'язаний з потоком **fs**, символьної змінної **ch** модна використати функцію **put ()**:

```
fs.put (ch);.
```

Приклади

1.
```
//  працюємо з файловими потоками в текстовому режимі - С++
#include <iostream>
#include <fstream>
using namespace std;
int main (int argc, char* argv [])
{
   ofstream _of ("text.txt");       // створюємо та відкриваємо
                                    // файловий потік для запису
   _of << "Hello, our dear world!\n"; // записали текст у файл
   _of.close ();                 // закрили файл із текстом
   char text [80];               // створили масив, в який читатимемо
                                 // текст із файлу
   ifstream _if ("text.txt");       // створюємо та відкриваємо
                                    // файловий потік для читання
     if (_if.is_open ())    // якщо файл успішно відкритий для читання
     {
```

```
        _if.getline (text, sizeof (text));   // читаємо відразу весь
                                             // рядок тексту
        cout << text << endl;                // виводимо прочитаний у масив
                                             // текст на екран
        _if.clear ();          // "чистимо" прапорці - якщо був
                               // досягнутий кінець файлу
        _if.seekg (0, ios::beg);  // встановлюємо стрілку файлу на
                                  // його початок
        // зараз читатимемо повторно по словах
        string word;
        while (_if >> word)    // поки можливо, читаємо слова
            cout << word << " "; // виводимо слово на екран
        cout << endl;
       _if.clear ();           // "чистимо" прапорці - адже був
                               // досягнутий кінець файлу
        _if.seekg (0, ios::beg);  // встановлюємо стрілку файлу на
                                  // його початок
        _if >> word;              // прочитали перше слово
        // зараз читатимемо ще раз повторно по словах,
        // з іншим способом контролю введення
        while (!_if.eof())    // поки не досягнутий кінець файлу
        {
         cout << word << " ";     // виводимо слово на екран
         _if >> word;             // читаємо чергове слово
        }
        cout << endl;
        _if.close ();         // закрили файл
    }
    else cout << "You have a problem with openinig file...\n";
  return 0;
}
```

Зауваження. В прикладі використовується одна з функцій так званого прямого доступу до файлу **seekg**. (Прямий доступ до файлів в стилі класичної мови С обговорювався в лекції 7). Отже, для організації прямого доступу до файлу використовують функції **seekg/seekp** та **tellg/tellp**. Різниця між ними в тому, що функції з іменем, що закінчуються на символ **'g'**, використовуються для роботи із потоками введення (скорочення слова **get**), а функції, з іменем, що закінчуються на символ **'p'**, – для роботи з потоками виводу (скорочення слова **put**).

Функції **seekg/seekp** переміщують внутрішній вказівник файлу (інколи кажуть стрілку файлу) на задану позицію. Позиції відповідають байтам, нумерація яких починається з **0**. Існує по 2 різновиди перевантажених функцій – з одним параметром та з двома параметрами. Якщо використовується один цілий параметр, то він задає абсолютную позицію у файлі. Якщо використовуються два параметри, то вони задають зсувс(у вигляді цілого число) та точку відліку. Точка відліку може набувати таких значень:

- **ios::beg** – початок файлу;
- **ios::cur** – поточна позиція вказівника;
- **ios::end** – кінець файлу.

Функції **tellg/tellp** не мають параметрів. Вони повертають поточну позицію вказівника у файлі.

Зверніть також увагу на вираз **(_if >> word)** – його значення ненульове допоки із файлу вдається прочитати об'єкт, це ознака успішності введення.

Результат виконання даного прикладу такий (один і той самий текст читається тричі різними способами):

```
Hello, our dear world!
Hello, our dear world!
Hello, our dear world!
```

3. Введення та виведення інформації у двійковому форматі.

У двійковому режимі символи у вигляді байтів, які рухаються потоками, не інтерпретуються. Для встановлення двійкового режиму, як вже вказувалось, треба задати прапорець **ios::binary**. Функція **get (ch)** дозволяє вводити окремий байт, а функція **put (ch)** – відповідно виводити. Для введення-виведення блоків інформації з двійкових файлів використовуються функції **read** и **write**. В ролі параметрів функції одержують вказівник (типу **char*** для функції **read** та типу **const char*** для функції **write**), який задає адресу початку масиву байтів для введення/виведення, та ціле число – кількість байтів для введення/виведення.

Приклади

2.
```
// працюємо із бінарними файловими потоками
#include <iostream>
#include <fstream>
using namespace std;

struct My   // струкутра містить різнорідну інформацію
{
    int i;
    float f;
    string s;
};
int main (int argc, char* argv [])
{
   My m1 = {123, 2.718281828, "test"};  // створили і заповнили
                                        // екземпляр структури
   My m2;                              // створили ще одну структуру
   fstream f;                          // створили екземпляр потоку
   f.open ("test.bin", ios::out | ios::binary);   // відкрили
                                       // двійковий файл для запису
   if (f.is_open ())                   // якщо файл успішно відкритий
   {
          // записуємо блок байтів структури
       f.write ((const char*)&m1, sizeof (m1));  // обов'язкове
                                  // приведення до типу const char*
       f.close ();    // закрили файловий потік
       f.open ("test.bin", ios::in | ios::binary);   // відкрили
                                       // двійковий файл для читання
```

```
        f.read ((char*)&m2, sizeof (m2));          // читаємо блок
                        // кодованої інформації з файлу в іншу структуру
        cout << m2.i << " " << m2.f << " " << m2.s << endl; // виводимо
                    // на екран для порівняння із початковою інформацією
    }
    return 0;
}
```

Зауваження. В цьому прикладі створений та проініціалізований екземпляр структури **m1**, який виводиться цілком як блок байтів розміру, рівного об'єму структури у двійковий файл. Якщо цей файл переглянути довільним засобом текстового перегляду, можна переконатись що інформація перебуває у кодованому вигляді. Потім цей самий файл відкривається повторно для введення і записана інформація блоком такого ж самого розміру зчитується в іншу структуру **m2**. Остання виводиться на екран для порівняння із вмістом структури **m1**. Результат порівняння – співпадіння вмісту обох структур.

Результат виконання цього прикладу такий:

```
123 2.71828 test
```

Контрольні питання.

1. Що таке потік?
2. Які є засоби для керування станом потоку?
3. Які функції контролюють стан потоку?
4. Які маніпулятори забезпечують форматування текстових потоків?
5. Які функції забезпечують прямий доступ до файлів?
6. Які засоби введення та виведення текстової інформації вам відомі?
7. Які засоби введення та виведення двійкової інформації вам відомі?

Завдання для самопідготовки.

1. Написати фрагмент коду, в якому потік **f** класу **ofstream** зв'язується з файлом **data.txt** і файл відкривається для: а) виведення в текстовому режимі з доповненням даних, що є у файлі; б) виведення в двійковому режимі зі знищенням даних, що є у файлі; в) виведення в двійковому режимі з доповненням даних, що є у файлі.
2. Написати фрагмент коду, в якому потік **f** класу **fstream** зв'язується з файлом **data.txt** і файл відкривається для: а) введення в текстовому режимі; б) введення в двійковому режимі.

Лекція 15. Шаблони функції та класів. Деякі контейнерні класи.

1. Що таке шаблони і для чого вони призначені.

У багатьох випадках перед програмістом виникає задача програмування деякого «універсального» алгоритму (універсального в тому сенсі, що його виконання не залежить від конкретних типів об'єктів, якими він оперує). Наприклад, при пошуку максимального елементу масиву ми діємо однаково, незалежно від того, чи є елементи даного масиву цілими, дійсними чи навіть символьними. Вирішальним при переході до наступного кроку є просто результат порівняння елементів. Іншими прикладами таких алгоритмів є сортування масиву чи побудова динамічної структури даних на кшталт стеку, зв'язаного списку або дерева. І тут постає питання: як врахувати можливість реалізації даного алгоритму для різних типів даних? Адже C++, як і багато інших мов, є типізованою мовою.

Для реалізації подібних алгоритмів програміст може обрати один із наступних шляхів:

- ✓ реалізувати один і той самий алгоритм для кожного потрібного типу даних;
- ✓ використати абстрактний базовий клас, в якому визначити даний алгоритм і заміщати його відповідним чином у похідних класах;
- ✓ використати засоби препроцесора і створити макровизначення;
- ✓ створити ***шаблон***, який конкретизується в залежності від типів параметрів.

Спробуйте обдумати вади та переваги кожного із відомих вам шляхів. Ми ж переходимо до вивчення і використання шаблонів функцій та класів.

Отже, ***шаблони*** призначені саме для розв'язання подібних задач, і оскільки вони є вбудованими засобами мови, для них забезпечується належна підтримка контролю типів.

Шаблон являє собою ***функцію*** або ***клас***, що можуть бути реалізовані для одного чи кількох абстрактних типів даних, які ***невідомі на момент компіляції цього коду***. В момент виклику у шаблон передаються ***конкретні типи даних***, для яких генерується функція чи клас із відповідними типами даних.

З іншого боку варто розуміти, що в результаті ми приходимо до певної філософської загрози, адже таким чином виникають програми, які створюють інші програми – ми маємо справу із так званим ***метапрограмуванням***.

2. Шаблон функції.

Шаблон функції (її ще називають ***родовою функцією***) – це узагальнене визначення цілого сімейства функцій, які можуть бути викликані ***з аргументами різних типів***.

Визначення шаблону починається із службового слова **`template`**, після якого у кутових дужках має знаходитись непорожній список параметрів шаблону.

Для визначенні параметрів шаблону, якими є ***типи*** параметрів даної функції, використовується службове слово **`class`**, яке втім не має жодного відношення до поняття класу в мові C++, або службове слово **`typename`**, затверджене дещо пізніше у стандарті мови. Після нього вказується довільний ідентифікатор параметру шаблону.

Загальний синтаксис визначення шаблону функції з одним параметром-типом такий:

```
template <typename T>
<тип результату> <ідентифікатор функції> (<параметри функції>)
{
     // тіло функції
}
```

Тут параметр **`T`** визначає тип параметру (параметрів), які є у списку параметрів функції. Ідентифікатор **`T`** тут вибраний довільно, звісно можна використати будь-який інший.

Розглянемо для прикладу функцію, яка повертає максимум з двох своїх параметрів.

Приклади

1.
```
// Знайомимся із шаблоном функції
#include <iostream>
using namespace std;
     // Шаблон функції, Type - ідентифікатор типу, який є параметром
     // цього шаблону функції
template <typename Type>
Type maxi (Type a, Type b)
{
     return (a < b)? b : a;
}
```

```
int main( )
{
 int n1 = 10, n2 = -20;
 double x1 = 1.0e-2, x2 = 1.0e+2;
 char c1 = 'A', c2 = 'B';
     // функція-шаблон maxi викликається для різних типів аргументів:
 cout << "Type = integer, maximum: " << maxi (n1, n2) << endl;
 cout << "Type = double,  maximum: " << maxi (x1, x2) << endl;
 cout << "Type = char,    maximum: " << maxi (c1, c2) << endl;
 return 0;
}
```

З'ясуємо, що відбувається при виклику функції-шаблону. При першому виклику **maxi (n1, n2)** системі стають відомими типи аргументів функції – вони обидва мають тип **int**. Тому ***генерується*** код функції **maxi** із аргументами **a** та **b** типу **int**. Ця функція викликається, відпрацьовує та повертає результат того ж типу **int**, який і виводиться на екран. Цей процес генерації коду функції із відповідними виклику типами параметрів називається ***конкретизацією*** (**instantiation**) шаблону функції. Таким чином, під час виконання вже відкомпільованої програми вона зазнає «підгонки під тип», одержуючи доповнення у вигляді коду конкретизованої шаблонної функції. Подібна процедура відбувається і при другому, і при третьому викликах з тою різницею, що генеруються інші варіанти шаблонної функції – відповідно із аргументами типів **double** та **char**. Повний результат виконання цього прикладу наведений нижче:

```
Type = integer, maximum: 10
Type = double,  maximum: 100
Type = char,    maximum: B
```

Слід зауважити, що крім параметрів-типів шаблони можуть містити і звичайні параметри, як у наступному прикладі. В ньому функція-шаблон порівнює два масиви однакової довжини. Параметрами цього шаблону є тип елементів порівнюваних масивів **T** та кількість елементів масиву **size**. В такому випадку конкретизація шаблону буде відбуватись не тільки в залежності від типу елементів масиву, але й в залежності від кількості цих елементів. Ця функція повертає логічний результат порівнння **true** або **false**.

Приклади

2.
```
// шаблонна функція для порівняння двох масивів
#include <iostream>
using namespace std;
template <typename T, int size>   // T - тип елементів масивів;
          // size - кількість елементів
bool eq_Array (const T* A, const T* B) // A та B – константні
          // вказівники на масиви, що порівнюються
{
     for (int i = 0; i < size; i++)
         if (A[i] != B [i]) return false;
     return true;
}

int main ()
{
     int A[] = {1, 2, 3};      // задаємо масиви для порівняння
```

```
    int B[] = {1, 2, 3};
    char C[] = {'a', '1'};    // задаємо масив для порівняння
    char *D = new char [2];   // цей масив створений динамічно
    D [0] = 'a';   D [1] = '2'; // задамо його елементи
    cout << boolalpha; //маніпулятор для виведення логічних констант
    cout << eq_Array <int, 3> (A, B) << endl;
    cout << eq_Array <char, 2> (C, D) << endl;
    return 0;
}
```

Зауваження. Зверніть увагу, тут для виклику цієї шаблонної функції потрібно явно вказати параметри шаблону **<int, 3>** та **<char, 2>**. Результат виконання цього прикладу:

```
true
false
```

Важливо зазначити, що ***родові функції (шаблони)*** можуть перевантажуватись. Причому може існувати кілька шаблонів функцій з різними наборами параметрів, а може також бути створена звичайна функція з іменем шаблона.

В останньому випадку перевантажена функція може перекривати («затіняти») шаблонну функцію, яку компілятор створив би для даного конкретного виклику. Крім того, перевантаження шаблонів може привести до складних і неоднозначних виборів функцій-кандидатів. Отже, можливість перевантаження шаблонів слід використовувати вкрай обережно.

Ще одну, вкрай важливу, обставину слід мати на увазі: параметром шаблону функції може бути ***будь-який тип***, зокрема і тип користувача, тобто клас. Синтаксично це дозволено. Проте, використовуючи подібний тип в якості параметра шаблона функції, маєте бути впевненими, що операції, які використовуються в цьому шаблоні функції, реалізовані і у вашому класі. Наприклад, виклик функції **maxi** не буде вдалим для параметру типу **Account** або **Figure_3D**, або й низки інших класів, які розглядались вище. Просто тому, що в них не реалізована операція порівняння **<** (перевірка на «менше»).

3. Шаблон класу.

Шаблон класу (або ***родовий клас***) – як і у випадку родової функції містить деякі універсальні алгоритми обробки даних параметричного типу, а конкретні типи даних підставляються в момент створення екземпляру даного класу. Таким чином, шаблон класу породжує ціле сімейство класів із спільною логікою функціонування для різних типів даних.

Синтаксис визначення шаблону класу з параметром **Type** наступний (замість службового слова **class** в кутових дужках можна використовувати службове слово **typename**):

```
template  <class Type> class class_id
{
    // тіло класу
};
```

Функції-члени класу автоматично стають шаблонами функцій, хоча для них і не вказується явно службове слово **template**. У випадку, коли така функція лише декларується у класі, а визначається поза ним, тоді для її визначення використовується достатньо складний синтаксис:

```
template  <class Type> тип_результату_функції
class_id <Type> :: func_id (параметри_функції)
{
    // тіло функції
```

```
}
```
При створенні екземпляру шаблону класу конкретні значення аргументів шаблону вказуються в кутових дужках:

```
class_id <int> c_i;
class_id <double> c_d;
```

Для даного прикладу під час виконання програми буде сгенеровано екземпляр **c_i** класу **class_id**, в якому в ролі типу **Type** виступатиме тип **int**, та екземпляр **c_d** того самого класу **class_id**, але із типом **double** замість типу **Type**.

Якщо шаблон має не один параметр, а більше, значення аргументів шаблону при створенні екземпляру мають однозначно відповідати списку параметрів шаблону.

В наступному прикладі створений шаблон класу, який є оболонкою універсального інкапсульованого масиву з елементами, тип яких є параметром шаблону (подібний клас, але з елементами фіксованого типу **int** був створений в лекції 11). В класі визначені конструктор, конструктор копіювання (пам'ятаємо, що для класів, полями яких є вказівники, конструктор копіювання реально необхідний), деструктор, перевантажений оператор-індексатор та є метод, що дозволяє задавати значення елементам інкапсульованого в класі масиву.

Приклади

3.
```
// шаблон класу – реалізація інкапсульованого масиву
#include <iostream>
using namespace std;
template <typename T>  // заголовок шаблону; T - тип елементів масиву
class buffer
{
  private:
     T* p;                 // вказівник на масив
  public:
     int n;                // кількість елементів масиву
     buffer (int n_);      //  конструктор
     buffer (const buffer& A);     //  конструктор копіювання
     ~buffer ();           // деструктор
     T operator [] (int ind); // перевантажений оператор доступу по
                           // індексу, тобто індексатор
     void set (int ind, T elem);   // встановлює значення елемента
                           // з заданим індексом
};
// визначаємо функції-члени шаблону класу
template <typename T> buffer <T> :: buffer (int n_)// конструктор
{
     n = n_;
     p = new T [n];
     cout << "Create array, size = " << n << " sizeof element = " <<
sizeof (T) << endl;
}
//  конструктор копіювання:
template <typename T> buffer <T> :: buffer (const buffer& A)
{
     n = A.n;                  // переписали кількість елементів
     p = new T [n];            // створили новий масив для копії
```

```
     cout << "Create copy array, size = " << n << " sizeof element =
" << sizeof (T) << endl;
     for (int i = 0; i < n; i++)
     p[i] = A.p[i];            // скопіювали елементи
}
template <typename T> buffer <T> :: ~buffer () // деструктор
{
     delete [] p;
     cout << "Destruct array, size = " << n << " sizeof element = "
<< sizeof (T) << endl;
}
// індексатор:
template <typename T>  T buffer <T> :: operator [] (int ind)
{
// перевіряємо правильність індексу
     if ((ind >= 0) && (ind < n))  return p[ind];
     else
     {
     cout << "Error: index out of range\n";
     return p[0];
     }
}
// встановлює значення елемента із заданим індексом:
template <typename T>  void buffer <T> :: set (int ind, T elem)
{
// перевіряємо правильність індексу
     if ((ind >= 0) && (ind < n)) p[ind] = elem;
     else cout << "Error: index out of range\n";
}
int main ()
{
     int A[] = {1, 2, 3, 4, -5};    // створили деякий масив
     char B[] = {'a', '1', '!'};    // створили деякий масив
     buffer <int> b1 (5);           // створили екземпляр шаблону
                                    // з цілочисельним масивом
     buffer <char> b2 (3);          // створили екземпляр шаблону
                                    // з символьним масивом
     for (int i = 0; i < b1.n; i++)
          b1.set(i, A[i]);          // записали елементи шаблону b1
     for (int i = 0; i < b2.n; i++)
          b2.set(i, B[i]);          // записали елементи шаблону b2
     for (int i = 0; i < b1.n; i++)
          cout << b1[i] << "\t";    // вивели елементи шаблону b1
     cout << endl;
     for (int i = 0; i < b2.n; i++)
          cout << b2[i] << "\t";    // вивели елементи шаблону b2
     cout << endl;
     buffer <int> b3 (b1); // тут спрацює конструктор копіювання
     return 0;
```

```
}
```

Зауваження. Зверніть увагу, тут створено два екземпляри шаблонного класу **buffer** : один з них **b1** для реалізації масиву з 5 елементів цілого типу **int**, а другий – **b2** для масиву з 3 елементів символьного типу **char**. Елементам присвоєні значення відповідно елементів масивів **A** та **B**. Крім того, для ілюстрації роботи конструктора копіювання створений екземпляр **b3**, як копія екземпляра **b1**. Нижче результат виконання цього прикладу.

```
Create array, size = 5 sizeof element = 4
Create array, size = 3 sizeof element = 1
1       2       3       4       -5
a       1       !
Create copy array, size = 5 sizeof element = 4
Destruct array, size = 5 sizeof element = 4
Destruct array, size = 3 sizeof element = 1
Destruct array, size = 5 sizeof element = 4
```

Зазначимо деякі правила визначення шаблонів функцій та класів:

- ✓ Шаблони функцій не можуть бути віртуальними.
- ✓ Шаблони класів можуть містити статичні елементи, дружні функції та класи.
- ✓ Шаблони класів можуть бути похідними як від звичайних класів, так і від шаблонів, а також бути базовими класами і для шаблонів, і для звичайних класів.

4. Бібліотека стандартних шаблонів мови С++.

В розпорядженні користувачів потужна бібліотека стандартних шаблонів (**STL – Standard Template Library**), яка містить ряд шаблонів класів. Ядро бібліотеки стандартних шаблонів складають 3 основні елементи:

- ✓ контейнери
- ✓ алгоритми
- ✓ ітератори.

Контейнер, як ми бачимо, є одним з елементів бібліотеки стандартних шаблонів, призначений для збереження даних. Контейнерами називають об'єкти, які містять інші об'єкти. Контейнери дозволяють керувати розміщенням об'єктів в оперативній пам'яті за допомогою конструкторів, та видаляти об'єкти з пам'яті за допомогою деструкторів. Для них реалізовані операції додавання та видалення об'єктів завдяки операціям вставки й видалення. Ітератори – це об’єкти, які дозволяють перебирати об’єкти в контейнері. Спрощено можна уявляти собі їх як вказівники на елементи контейнеру. Від вказівників їх відрізняє потужний функціонал, зокрема ряд перевантажених операцій. Алгоритми реалізують певний набір задач для об’єктів у контейнері, як то: впорядкування, сортування, тощо.

Деякі контейнерні класи наведені у таблиці нижче.

Контейнер	Опис	Заголовок
bitset	множина бітів	**<bitset>**
queue	черга	**<queue>**
list	лінійний список	**<list>**

set	множина, в якій кожний елемент унікальний	**<set>**
stack	стек	**<stack>**
vector	вектор	**<vector>**
map	асоційований список для збереження пар ***ключ-значення***	**<map>**

Ця таблиця містить контейнери двох видів – послідовні та асоціативні. У послідовних контейнерах скінчена множина їх елементів (об'єктів) розташовані впорядковано, в асоціативних контейнерах головною метою використання є швидкий пошук об'єкта по ключу. У всіх контейнерів є ряд однакових операторів, допоміжних фуекцій, перевантажені операції порівняння та присвоєння.

В якості ілюстрації розглянемо приклад використання контейнерного класу **vector**, який є фактично динамічним масивом з довільним типом елементів (цей тип і є параметром шаблону класу **vector**). Цей клас окрім довільного доступу до своїх елементів дозволяє також вставку та видалення елементів, автоматичне збільшення розміру, пересування ітераторів по елементам.

Приклади

4.
```
// Знайомство з шаблоном класу vector
#include <iostream>
#include <vector>   // підключили файл заголовків контейнеру vector
using namespace std;

template <typename T>     // шаблон корисної функції виводу на екран
                          // вектора v з типом елементів T
void print(vector<T> &v)
{
      if (v.empty())      // якщо вектор порожній - повідомимо про це
      {
            cout << "vector is empty\n";
      }
      else
      {
            typename vector<T>::iterator it;   // задаємо ітератор it
            // ітератор пробігає всі елементи вектора
            for (it = v.begin(); it != v.end(); it++)
                  // виводимо на екран елемент, на який вказує ітератор
                  cout << *it << "\t";
            cout << endl;
      }
}

int main ()
```

```
{
     vector <char> v1;    // створили порожній вектор символів
     v1.push_back('A');   // вставили в нього символ A
     v1.push_back('B');   // вставили в кінець ще один символ B
     cout <<"v1_max_size = "<< v1.max_size() << "   v1_size = " <<
v1.size() << endl;
     print (v1);          // вивели вектор символів на екран
     vector <double> v2; // створили порожній вектор дійсних чисел
     cout <<"v2_max_size = "<< v2.max_size() << "   v2_size = " <<
v2.size() << endl;
     v2.insert(v2.begin(), 3, 1.5);// додали три елементи, рівних
                                   // 1.5 з початку вектора
     print (v2);          // вивели вектор дійсних чисел на екран
     v2.insert(++v2.begin(), 2, 2.5);   // додали два елементи,
                                   // рівних 2.5 після 1-го елемента
     print (v2);          // вивели вектор дійсних чисел на екран
     v2.pop_back();       // видалення останнього елементу
     print (v2);          // вивели вектор дійсних чисел на екран
     cout <<"v2_max_size = "<< v2.max_size() << "   v2_size = " <<
v2.size() << endl;
//   vector <int> v2 = {1, 2, 3}; // це дозволяє не кожний компілятор,
          // проте, можна ініціювати новостворений вектор масивом
     int array [] = {1, 2, 3};
     vector <int> v3 (array, array + 3);// створили вектор цілих
                          // чисел та ініціювали його масивом array
     v3.push_back(100);   // вставили в кінець вектора число 100
     v3.at (0) = 400;     // змінили елемент з індексом 0 на 400
     print (v3);          // вивели вектор цілих чисел на екран
     v3.erase (v3.begin() + 1); // видалили з вектора елемент, що є
                                // першим (після 0-го)
     print (v3);          // вивели вектор цілих чисел на екран
     vector <int> v4 = v3;// створили копію вектора v3
     v3.clear();          // вичистили вектор v3
     print (v3);          // спроба вивести на екран порожній вектор
     print (v4);// проте вектор v4 зберігає останній стан вектора v3
     return 0;
}
```

Зауваження. Зверніть увагу: функція-член контейнеру **max_size** повертає максимально можливий об'єм пам'яті, виділений під даний об'єкт, функція-член контейнеру **size** повертає кількість елементів в даному векторі в поточному його стані. Проаналізуйте код шаблону функції **print**, параметром якої є вектор з довільним типом елементів, тому ця функція справляється із виводом на екран всіх трьох векторів з різними типами елементів. В цьому прикладі використані деякі функції класу **vector**, які легко дозволяють змінювати зміст конкретного екземпляру вектора. Ідентифікатори цих функцій красномовно свідчать про їх призначення, додаткова інформація міститься в коментарях до кожного рядка. Результат виконання цього прикладу такий:

```
v1_max_size = 18446744073709551615  v1_size = 2
A       B
v2_max_size = 2305843009213693951  v2_size = 0
```

```
1.5     1.5     1.5
1.5     2.5     2.5     1.5     1.5
1.5     2.5     2.5     1.5
v2_max_size = 2305843009213693951  v2_size = 4
400     2       3       100
400     3       100
vector is empty
400     3       100
```

Контрольні питання.

1. Яка мета створення та використання шаблонів?
2. Що таке шаблон функції?
3. Що таке конкретизація шаблону функція та як вона відбувається?
4. Які параметри можуть бути у шаблона функції?
5. Що таке шаблон класу?
6. Що таке **STL**?
7. Що таке контейнерний клас?

Завдання для самопідготовки.

1. Створіть шаблон функції для порівняння двох параметрів різних типів. Ці типи задайте параметрами шаблону.
2. Напишіть приклад з використанням одного з контейнерів. Відпрацюйте наявні в цьому контейнері функції.

Література

1. Основи програмування. Мова C#. Методичний посібник для студентів радіофізичного факультету університету. / В.О. Грязнова, С.В.Єфіменко, К.Е.Юштін – К.: КНУ, 2009
2. Язык программирования C. Второе издание, переработанное и дополненное/ Керниган Б., Ритчи Д.
3. C++ Базовый курс. /Герберт Шилдт.

Зміст

Навчальне видання

ЄФІМЕНКО *Світлана Володимирівна*

Методичний посібник з курсу «Об’єктно-орієнтоване програмування. Мови С/С++»