ПРИВАТНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«КОМП'ЮТЕРНА АКАДЕМІЯ ШАГ»

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни «**Вступ до програмування ч.1 (С)**»

галузь знань **12 Інформаційні технології**

спеціальність **122 Комп'ютерні науки**

Львів - 2017

Конспект лекцій з навчальної дисципліни «**Вступ до програмування ч.1 (C/C++)**» для студентів за спеціальністю 122 Комп'ютерні науки

розробник:
професор, д.т.н., професор кафедри _____

ЗМІСТ

Лекція 1. Загальна характеристика мови С. Склад мови. Структура проектів С++. Коментарі, вирази і l-вирази.

1.1. Загальна характеристика мови С++

С++ являє собою високоорганізовану систему програмування, яка повністю підтримує Standart С++ згідно з специфікацією ANSI/ISO. Принциповими відмінностями С++ від ANSI C є:

1) підтримка об'єктно-орієнтованого (класи, class) та узагальненого (шаблони, template) програмування;
2) можливість перевантаження функцій і операторів (overloading);
3) необов'язковість зарезервованого слова void в оголошенні функції з порожнім переліком параметрів;
4) збільшення обов'язкове оголошення прототипів функцій;
5) обов'язкове використання в тілі функції, яка повертає відмінне від void значення, оператора return зі значенням даного типу;
6) оголошення локальних змінних у будь-якому місці програми до моменту їх першого використання;
7) введення булевого типу bool і зарезервованих слів true і false;
8) оператори управління динамічною пам'яттю та системою потоків вводу/виводу;
9) розширення різновидів функцій;
10) розміщення ідентифікаторів структур, класів та об'єднань у глобальному просторі імен, що призвело з одного боку до скорочення синтаксису при створенні об'єктів цих типів, а з іншого виключило конструкції, які дозволяли створення змінних ти користувацьких типів в одній області видимості.
11) концептуальне розширення можливостей та спрощення оголошень деяких типів, зокрема структур та об'єднань;
12) поява підсистеми генерації та обробки виключень (try, throw, catch), технологій динамічної ідентифікації типів (RTTI) та просторів імен (namespace);
13) поява нових типів та нового формату оператора приведення типу.

Запропонований список не претендує на повноту, а відображає лише основні, з точки зору авторів, відмінності між мовами ANSI C і C++ .

Мова С++ розширює список зарезервованих слів ANSI C. Як будь-мова, С++ постійно розвивається і нововведення поступово стандартизуються.

При написанні стандарту С++11 (робоча назва С++0x) розробники керувались такими правилами:

- підтримка стабільності мови при забезпеченні сумісності із стандартом С++98 і, при можливості, із мовою С;
- реалізація нових можливостей мови повинна здійснюватися засобами бібліотек, а не через ядро мови;

- модифікація окремих семантичних і синтаксичних конструкцій, які покращають техніку програмування, спрощують вивчення мови без видалення можливостей, що використовуються експертами в області програмування;
- удосконалення C++ з позицій системного і бібліотечного наповнення заміть додавання можливостей, які є корисними для розробки окремих програм;
- підвищення рівня стійкості та безпеки при роботі з типами;
- збільшення продуктивності і можливостей при розробки як вбудованих систем (embedded systems) так і систем прямої взаємодії з апаратним забезпеченням;
- забезпечення розв'язання реальних, широко розповсюджених проблем;
- реалізація принципу "оплата лише за те, що використовується".

Вважається, що C++ є розвитком мови C. Незважаючи на те, що більша частина коду C підтримується у C++, мова C не є підмножиною C++. Це означає, що C++ не включає C повністю. Існує програмний код, який вірний для C, але не є валідним для C++. Типовим ілюстративним прикладом є виклик функції main() усередині програми. У мові C такий виклик допускається. У мові C++ така дія є неправомірною. Більше того, в окремих синтаксичних конструкція C++ стала більш строгішою, наприклад не допускається неявне приведення типів для вказівників на різний тип і ін. Загалом навіть результуючий код компіляторами мов C і C++ може генеруватись різний. Наприклад у наступному прикладі різне трактування типу char компіляторами мов C і C++ приведе до появи на екрані різних повідомлень

Приклад № 1.1.

```
#include <stdio.h>
int main(void)
{
puts( (sizeof(char) == 1) ? "C++ compiler" : "C compiler");
}
```

Основними перевагами мови C++ є:

- у мові реалізована підтримка різних видів програмування, зокрема програмування з використанням директив, об'єктно-орієнтоване програмування, узагальнене програмування та метапрограмування;
- увесь код, який неявно генерується компілятором і призначений для реалізації можливостей мови є визначений стандартом. Це визначає передбачуваність поведінки коду і строгу визначеність точок виконання коду, що у свою чергу, є важливою перевагою на користь використання мови при побудові систем фактично будь-якого призначення, у тому числі реального часу, вбудовуваних систем, низькорівневих утиліт та драйверів;
- зворотній відносно порядку виклику конструкторів виклик деструкторів є автоматичним при знищенні об'єктів. Це спрощує і підвищує

надійність звільнення ресурсів, а також забезпечує гарантію виконання переходів станів програми, які необов'язково пов'язані із звільненням ресурсів;

- навантаження операторів у звичному для сприйняття виді дозволяють стисло і повно формувати операції стосовно користувацьких типів;
- підтримка за допомогою модифікаторів фізичної (const) і логічної (mutable) постійності спрощує компілятору діагностування помилок і є непрямою директивою для оптимізацію коду. Використання константних методів у класах дозволяє організувати навантаження, яке спрощує визначення із середини мети виклику (константний метод виключно для читання, не константний – для зміни). Використання модифікатора mutable дозволяє зберігати логічну незмінність при використанні кешу і відкладених обчислень;
- існують можливості:
  - імітації розширення мови для підтримки парадигм, які не підтримуються компіляторами безпосередньо;
  - створення вбудованих предметно-орієнтованих мов програмування. (наприклад бібліотека Boost.Spirit і формальна система визначення синтаксису - EBNF-граматика парсерів);
  - низькорівневої роботи з пам'яттю та адресами;
  - імітації класів-домішків і комбінаторної параметризації бібліотек (наприклад, бібліотека Loki, клас SmartPtr);
- завдяки тому, що стандарт мови накладає мінімальні вимоги на ЕОМ для запуску відкомпільованих програм, достатньо легко можна забезпечити кросплатформеність. Для визначення реальних властивостей середовища виконання в стандартній бібліотеці присутні відповідні можливості (наприклад, std::numeric_limits <T>);
- завдяки проектуванню мови так, щоб надати програмістові максимальний контроль над усіма аспектами структури та порядку виконання програми, забезпечується висока ефективність. Є необов’язковою будь-яка мовна конструкція, яка призводить додаткових обчислювальних витрат. У випадку необхідності C++ дозволяє забезпечити максимальну ефективність програми;
- висока сумісність з мовою C дозволяє використовувати напрацьований C-код.

Розглянемо деякі базові поняття і структури C++ , без правильного розуміння яких є неможливим досконале вивчення цієї мови.

1.2. Структура проектів C++

Проекти C++ представляють собою складну ієрархічну базу даних і за структурою дуже подібні до проектів ANSI C. Вони не визначені стандартом, а тому відрізняються від одного компілятора до іншого. Зазвичай у сучасних середовищах розробки об'єднання файлів у логічну структуру здійснюються за допомогою базового файлу у форматі xml (наприклад у MS Visual Studio таким є файл з розширенням vcxproj).

Оскільки структура проекту є залежного від виробника компілятора, то до складу проекту, зазвичай, входять файли різних типів і різноманітного призначення. Спільним для усіх проектів від різних проектів є файли з кодом C/C++. Стандарт, у якості рекомендації, визначає два види файлів, які повинні входити до складу проекту і містити код C/C++ — це файли коду (ФК, або компільовані файли, тобто файли, які компілюються у файли незв'язаного (nonlinked) коду — obj- файли. Рекомендоване розширення c і cpp) та файли заголовків (ФЗ, або некомпільовані файли. Рекомендоване розширення h і hpp).

Основним використанням ФЗ є його включення спеціальною програмою препроцесором на попередньому етапі (перед компіляцією) у точці підмикання (включення, директива #include) ФК. Відповідно процес компіляції уже буде стосуватись не вихідного, а об'єднаного файлу. Сама процедура підмикання називається компонуванням, об'єднанням або препроцесоруванням.

Цільовим призначенням ФЗ є концентрація у одному файлів різноманітних оголошень, які не виступають виконавчим кодом (наприклад визначення типу є невиконавчим кодом. Але не визначення змінної, яке є операцією, а тому компілюється у виконавчі інструкції). Будь-яка із наступних інструкцій не може використовуватись у ФЗ

```
float f;
extern int i = 0;
extern void f1() {}
void f2() {}
```

У перших двох рядках, незалежно від використання зарезервованого слова extern, здійснюється операція створення глобальної змінної, що є виконавчими діями. Третій та четвертий рядки містять визначення тіла функції, що також вважаться виконавчими діями і суперечить ідеології ФЗ. Це зумовлено тим, що підмикання цього ФЗ у двох чи більше файлах коду спричинить неоднозначність зв'язування.

Розглянемо такий ФЗ

Файл - var.h

```
#ifndef VARH
#define VARH
	typedef unsigned int UI;
	const int i = 50;
	inline UI max(UI a, UI b)
	{
		return ( a > b ) ? a : b;
	}
#endif
```

Дисонансом у ньому виступають визначення константи та вбудованої функції. Проте ці оголошення не суперечать ідеї ФЗ, оскільки константи, які визначаються за допомогою модифікатора const, та inline-функції є спеціальними видами оголошень, які можуть в програмі виникати декілька разів. Проте рекомендується у ФЗ залишати лише декларацію константи, а її визначення виносити у файл коду, наприклад

Файл - var.h

```
#ifndef VARH
#define VARH
        extern const int i = 50;
#endif
```

Файл - var.cpp

```
#include "var.h"
const int i = 50;
```

З практичної точки зору ФЗ є бібліотекою оголошень зовнішніх (extern) об'єктів, функцій і визначення вбудовуваних функцій. Це називається локалізацією оголошень. Файли, які містять визначення об'єктів та функцій повинні підмикати ФЗ.

Підхід з використанням ФЗ дозволяє досягти двох цілей:

- гарантується єдність оголошення глобальної сутності в усіх файлах проекту;
- у випадку потреби зміни оголошення, вона проводиться один раз і цим гарантується її задіюваність у всіх файлах вихідного коду, які підмикають даний ФЗ.

При проектуванні ФЗ треба враховувати декілька моментів. Структури, класи та різноманітні декларації (наприклад декларації директиви typedef) рекомендується оголошуються у ФЗ.

Усі оголошення ФЗ повинні бути логічно зв'язаними. Якщо він є достатньо великим, або містить велику кількість не зв'язаних між собою декларацій, то велика імовірність відмови використання цього ФЗ розробником. І основним обґрунтуванням буде виступати зменшення часу компіляція.

Для визначенні імені ФЗ у випадку мови C++ пропонується використовувати розширення hpp. Це рекомендується для того, щоб підкреслити їх відмінність від файлів з розширенням h, яке, у свою чергу рекомендується стандартом ANSI C. Такий підхід обґрунтований тим, що ФЗ мови C++ можуть містити конструкції не сумісні з ANSI C. Подібний підхід рекомендується також для випадку ФК — при використанні мови C вихідний код повинен

розміщуватись у файлі з розширенням c, а при використанні мови C++ — у файлах з розширенням cpp.

Зазначимо, що описана схема використання розширень для іменування файлів та структура розміщення коду мають виключно рекомендаційний характер. Для визначення ФК і ФЗ можна використовувати будь-які розширення (наприклад поширеним є використання для ФК розширення cc). Подібно не забороняється винесення у файл із розширенням h виконавчих інструкцій. Але при цьому даний файл перестає бути ФЗ і компілятор його розглядає у якості ФК (тобто він стає компільованим).

Побудова багатофайлових проектів може призвести до ситуації коли компіляція буде тривалою в часі. Для того, щоб постійно не перекомпільовувати ФЗ проекту, особливо, якщо вони є дуже великими за розміром, можна використовувати їх прекомпіляційні версії, які зберігаються на диску. Це здійснюється вмиканням опції типу Precompiled Header (наприклад у середовищі BS5 у вікні Options|Project|+Compile|Precompiled header; у середовищі MSVS2010 у вікні ProjectName Property Pages|C/C++|Precompiled header, або ключі компіляції /Y *). Тоді компілятор проглядає усі ФЗ, які підмикаються у вихідному файлі і зберігає їх на диску у відповідному файлі (наприклад для BS5 це файл bcwdef.csm для MSVS2010 — це PCH-файли). Зазначимо, що попередня компіляція стосується будь-якого, у тому числі і вбудованого, коду C чи C++.

Сучасні середовищі через технологію прекомпільованих заголовків підмикають можуть підмикати набори системних бібліотек. Так підхід дуже використовується, наприклад у MSVS2010, через файл stdafx.h.

З метою використання загальнодоступних класів або змінних їх оголошують в ФЗ, які підмикається директивою #include в кожний модуль, де використовуються даний клас або змінна. У C++ існує два методи підмикання файлів:

a) #include <file.ext> — файл file.ext ідентифікується в каталогах вказаних у налаштуваннях IDE-середовища (наприклад для BS5 це Options|Project|Directories (поточний каталог не розглядається)).

b) #include "file.ext" — ідентифікація файлу відбувається в поточному каталозі і в разі його відсутності за варіантом a).

При вказуванні імен файлів, які підмикаються, регістр букв не відіграє значення. Сама директива #include набирається у нижньому регістрі. Пропуски в оголошені імені файла, який підмикається директивою, трактуються як невід'ємна частина імені файла.

З метою уникнення ризику багаторазового підмикання ФЗ в один і той же файл через інші використовують директиви препроцесора, наприклад

Поряд зі старим в C++ додано новий формат директиви #include. У цьому форматі вказується ім'я бібліотеки, а не ФЗ. Імена бібліотек можуть бути набагато довшими за імена ФЗ. Компілятор сам зв'язує ім'я бібліотеки з відповідним ФЗ, наприклад за новим форматом підмикання потокової

підсистеми вводу виводу можна записати так: #include <iostream>. Проте можна використовувати старий формат: #include <iostream.h> .

Використовувати новий формат директиви треба дуже обережно. Концептуально він призначення для підмикання ФЗ без розширення. У випадку відсутності останніх компілятор на початку спробує знайти і підімкнути ФЗ з розширенням *.h. Якщо такого ФЗ не знайдено, компілятор спробує підімкнути файл з розширенням *.hpp. Так, наприклад для компілятора BSB6, директива #include <string> спричинить підмикання ФЗ з контейнерним класом string, а не бібліотеки функцій для роботи зі стрінгами у форматі *.char ('\0'-стрінгами).

Зазначимо, що порядок підмикання різних файлів у новому форматі може відрізнятись для різних компіляторів.

1.3. Коментарі, вирази і l-вирази

У мові C++ використовується новий формат коментарів - стрічок коду, які повністю ігноруються компілятором і використовується для внесення пояснень в текст програми. Згідно з ним, коментарі повинні починатись з символів // і займати одну стрічку. Старий формат /* ... */ підтримується також. Допускається використання вкладених коментарів як композиція старого та нового форматів. Проте деякі компілятори для цього можуть потребувати включення додаткової опції (наприклад для компілятора BS5 це опція Nested Comments у вікні Options|Project|+Compile|Source). Побудова вкладених коментарів, записаних лише у старому форматі, забороняється.

У випадку, коли необхідно вилучити з розгляду частину коду, поряд з коментарями використовують директиви препроцесора

#if... #else... #endif

Доступ до об'єктів і функцій в C++ забезпечується виразами, які в цьому випадку посилаються на об'єкти.

Вирази, які повертають посилання на константу, змінну чи функцію називаються l-виразами. Значення, які повертають l-вирази, називають lvalue. Тобто у якості lvalue викликаються вирази, які ідентифікують елементи даних з правильно визначеними і програмно доступними адресами. Ім'я змінної в C++ є частковим випадком l-виразу.

Термін lvalue походить з виразу присвоєння $e_1 = e_2$, в якому лівий операнд е повинен бути адресним (value) виразом. Це означає, що результат виразу $e_1=e_2$ можна представити посиланням на фізичну існуючу (не тимчасову) ділянку пам'яті, яка була виділена під лівий операнд (тобто e_1). Тобто виконання інструкції $e_1 = e_2$ компілятором трактується як виклик функції operator=(e_1, e_2), сигнатура якої має вид int& operator = (int&, int&). Такий підхід дає можливість організовувати конструкції продовженого виду, наприклад $e_1+=e_2=e_3$, яка

забезпечується послідовним викликом двох операторів: operator+=(e_1, operator=(e_2, e_3)).

Значення lvalue треба розглядати в парі з rvalue, проте rvalue не є lvalue. Наприклад, вираз x + 2 володіє значенням і є rvalue, проте він не є lvalue. А от при трактуванні бінарного виразу $e_1 = e_2$ можна чітко розрізнити не лише lvalue (e_1), а й rvalue – e_2 . Наприклад, у записі

```
int x, y = 1;
x = y + 1;
```

результат додавання y + 1 повинен існувати перед копіювання у комірку змінної x (найімовірніше він буде зберігатись в регістрі). Це означає, що за умови існування, результат дії y+1 є правими (другими) операндом функції operator=(x,y + 1).

Вирази, які володіють добре визначеними значеннями викликаються як rvalue. Значення такого виду зазвичай з'являються в правій частині виразів з присвоєння. У прикладі $e_1 += e_2 = e_3$ результат присвоєння $e_2 = e_3$ є lvalue, яке рівне e_2. Це lvalue у якості rvalue передається оператору int& operator+= (int&, int&). Якщо наведений приклад модифікувати і у лівій частині покласти дужки - ($e_1=e_2+2$) = 5, то вираз $e_1 = e_2 + 2$ після присвоєння матиме результатом (lvalue) e_1 і адреса цієї змінної виступатиме лівим (першим) операндом та результат другої операції присвоєння. Якщо записати (e_2 + 2) = 5, то виконання дії є неможливим, оскільки неможливо ідентифікувати комірку пам'яті лівого операнда. Останнє зумовлене тим, що дія e_2 + 2 не повертає lvalue.

Принциповою відмінністю lvalue від rvalue є те, що перші завжди репрезентують змінні, які допускають модифікацію. Незважаючи на те, що змінні, які оголошуються з модифікатором const, не дозволяють змін, вони також є lvalue (у явній формі це має місце у випадку, коли модифікацію значення намагаються здійснити через ідентифікатор змінної, яка оголошена з модифікатором const. Тут треба пам'ятати, що таку дію можна здійснити використавши інший доступ до даної комірки пам'яті).

Через розширення поняття тимчасовий об'єкт стандарт C++11 модифікує використання rvalue.

У мові C++ допускається змінна значень змінних. Значення констант констант і функцій змінам не підлягають. Тому l-вираз називається модифікованим l-виразом, або ліводоступним виразом у тому випадку, коли він не посилається на функцію, масив чи константу. Приведемо перелік виразів, які трактуються в контексті l-виразів:

1) ідентифікатор nonconst змінної.
2) результат виклику функції, якщо остання повертає посилання.
3) оператор присвоєння у цілому (lvalue є значенням є лівої частини після виконання присвоєння);

4) вирази з префіксними, але не з постфіксними операторами інкременту та декременту (тобто: ++k повертає lvalue, а k++ — ні);

5) оператор розіменування вказівників (*p) або індексуванням елементів масиву (tab[k]);

6) оператор доступу -> , якщо він не посилається на метод класу;

7) тенарний оператор, якщо обидва значення є lvalue (наприклад, вираз a > 0 ? ++i : ++j повертає lvalue, а вираз a > 0 ? i++ : j++ - ні);

8) приведення до типу посилання ((int&)k);

9) оператор послідовного виклику ‘,', якщо правий аргумент є lvalue.

Наостанок стосовно поняття lvalue приведемо для самоаналізу такий програмний код

Приклад № 1.2.

```
#include <iostream>
using namespace std;
int& timestwo(int& m)
{
    static int count;
    cout << " In timestwo: count = " << count << endl;
    m *= 2;
    return count;
}
void printTab(int * tab, int size)
{
    cout << "[";
    for (int i = 0; i < size; cout << tab[i] << " ", ++i) ;
    cout << "]" << endl;
}

void main(void)
{
    int i = 1, j = 2, k = 3;
    i;                                          // return of l-value
    (i = j) = k;                                // assignment as an l-value
    cout << " A: i = " << i << " j = " << j
       << " k = " << k << endl; // 3,2,3
    int tab[] = {1, 2, 3, 4},
    *p = tab;
    cout << " B: before ";
    printTab(tab,4);
    *++++++p = 8;
    cout << " B: after ";
```

```
    printTab(tab,4);
    // now p points to the last element!
    cout << " C: ++*----p = " << ++*----p << endl; // 3
    cout << " C: tab ";
    printTab(tab,4);
    int m = 7;                                  // conversion as an l-value
    timestwo( (int&)m = 8 )++;                  // conversion unnecessary!
    cout << "D1: m = " << m << endl;
    timestwo(m)++;
    cout << "D2: m = " << m << endl;
    int n = timestwo(m) = 10;
    cout << "D3: m = " << m << endl;
    cout << "D4: n = " << n << endl;
    k = (i = 1, j = 2) + 1;                     // comma operator
    cout << " E: i = " << i << " j = " << j
       << " k = " << k << endl;                 // 1,2,3
    (k > 2 ? i : j) = 5;                    // conditional expression as an l-value
    cout << " F: i = " << i << " j = " << j
    << " k = " << k << endl;                    // 5,2,3
}

.
```

Лекція 2. Типи даних.

Основна мета будь-якої програми полягає в обробці даних. Дані різного типу зберігаються і обробляються по-різному. У будь-якій алгоритмічній мові кожна константа, змінна, результат обчислення виразу або функції повинні мати певний тип.

Тип даних визначає:

- внутрішнє представлення даних в пам'яті комп'ютера;
- безліч значень, які можуть приймати величини цього типу;
- операції і функції, які можна застосовувати до величин цього типу.

Виходячи з цих характеристик, програміст вибирає тип кожної величини, використовуваної в програмі для подання реальних об'єктів. Обов'язкове опис типу дозволяє компілятору проводити перевірку допустимості різних конструкцій програми. Від типу величини залежать машинні команди, які будуть використовуватися для обробки даних.

Всі типи мови C ++ можна розділити на основні і складові. У мові C ++ визначено шість основних типів даних для представлення цілих, дійсних, символьних і логічних величин. На основі цих типів програміст може вводити опис складових типів. До них відносяться масиви, перерахування, функції, структури, посилання, покажчики, об'єднання і класи.

Основні (стандартні) типи даних часто називають арифметичними, оскільки їх можна використовувати в арифметичних операціях. Для опису основних типів визначені наступні ключові слова:

- int (цілий);
- char (символьний);
- wchar_t (розширений символьний);
- bool (логічний);
- float (речовинний);
- double (речовинний з подвійною точністю).

Перші чотири типи називають цілочисельними (цілими), останні два - типами з плаваючою точкою. Код, який формує компілятор для обробки цілих величин, відрізняється від коду для величин з плаваючою точкою.

Існує чотири специфікатори типу, що уточнюють внутрішнє уявлення і діапазон значень стандартних типів:

- short (короткий);
- long (довгий);
- signed (знаковий);
- unsigned (беззнаковий).

2.1. Цілий тип (int)

Розмір типу int не визначається стандартом, а залежить від комп'ютера і компілятора. Для 16-розрядного процесора під величини цього типу відводиться 2 байти, для 32-розрядної - 4 байта.

Специфікатор short перед ім'ям типу вказує компілятору, що під число потрібно відвести 2 байта незалежно від розрядності процесора. Специфікатор long означає, що ціла величина буде займати 4 байти. Таким чином, на 16-разрядних комп'ютерах еквівалентність int і short int, а на 32-розрядному - int і long int.

Внутрішнє представлення величини цілого типу - ціле число в двійковому коді. При використанні специфікатор signed старший біт числа інтерпретується як знаковий (0 - додатне число, 1 – від'ємне). Специфікатор unsigned дозволяє представляти тільки додатні числа, оскільки старший розряд розглядається як частина коду числа. Таким чином, діапазон значень типу int залежить від специфікаторів. Діапазони значень величин цілого типу з різними специфікаторами для IBM PC-сумісних комп'ютерів наведені в табл. 2.1.

За замовчуванням всі цілочисельні типи вважаються знаковими, тобто специфікатор signed можна опускати.

Константам, яке трапляється в програмі, приписується той або інший тип відповідно до їх виду. Якщо цей тип з яких-небудь причин не влаштовує програміста, він може явно вказати необхідний тип за допомогою суфіксів L, l (long) і U, u (unsigned). Наприклад, константа 32L матиме тип long і займати 4 байти. Можна використовувати суфікси L і U одночасно, наприклад, 0x22UL або 05Lu.

2.2. Символьний тип (char)

Під величину символьного типу відводиться кількість байт, достатню для розміщення будь-якого символу з набору символів для даного комп'ютера, що і зумовило назву типу. Як правило, це 1 байт. Тип char, як і інші цілі типи, може бути зі знаком або без знаку. У величинах зі знаком можна зберігати значення в діапазоні від -128 до 127. При використанні специфікатор unsigned значення можуть знаходитися в межах від 0 до 255. Зтого досить для зберігання будь-якого символу з 256-символьного набору ASCII. Величини типу char застосовуються також для зберігання цілих чисел, що не перевищують межі зазначених діапазонів.

2.3. Розширений символьний тип (wchar_t)

Тип wchar_t призначений для роботи з набором символів, для кодування яких недостатньо 1 байта, наприклад, Unicode. Розмір цього типу залежить від реалізації; як правило, він відповідає типу short. Строкові константи типу wchar_t записуються з префіксом L, наприклад, L "Gates".

2.4. Логічний тип (bool)

Величини логічного типу можуть приймати тільки значення true і false є зарезервованими словами. Внутрішня форма подання значення false - 0 (нуль). Будь-яке інше значення інтерпретується як true. При перетворенні до цілого типу true має значення 1.

2.5. Типи з плаваючою точкою (float, double і long double)

Стандарт C ++ визначає три типи даних для зберігання речових значень: float, double і long double.

Типи даних з плаваючою точкою зберігаються в пам'яті комп'ютера інакше, ніж цілочисельні. Внутрішнє представлення дійсного числа складається з двох частин - мантиси і порядку. В IBM PC-сумісних комп'ютерах величини типу float займають 4 байта, з яких один двійковий розряд відводиться під знак мантиси, 8 розрядів під порядок і 23 під мантису. Мантиса - це число, більше 1.0, але менше 2.0. Оскільки старша цифра мантиси завжди дорівнює 1, вона не зберігається.

Для величин типу double, що займають 8 байт, під порядок і мантису відводиться 11 і 52 розряду відповідно. Довжина мантиси визначає точність числа, а довжина близько - його діапазон. Як можна бачити з табл. 2.1, при однаковій кількості байт, що відводиться під величини типу float і long int, діапазони їх допустимих значень сильно розрізняються через внутрішню форми подання.

Специфікатор long перед ім'ям типу double вказує, що під величину відводиться 10 байт.

Константи з плаваючою точкою мають за замовчуванням тип double. Можна явно вказати тип константи за допомогою суфіксів F, f (float) і L, l (long). Напри заходів, константа 2E + 6L матиме тип 1ong doub1e, а константа 1.82f -тип float.

Таблиця 2.1. Діапазони значень простих типів даних для IBM PC

Тип	Діапазон значень	Розмір (байт)
bool	true і false	1
signed char	-128 ... 127	1
unsigned char	0 ... 255	1
signed short int	-32 768 ... 32 767	2
unsigned short int	0 ... 65 535	2
signed long int	-2 147 483 648, ... 2 147 483 647	4
unsigncd long int	0, ... 4 294 967 295	4
float	3.4e-38 ... 3.4e + 38	4
double	1.7e-308 ... 1.7e + 308	8

long doublc	3.4e-4932 ... 3.4e + 4932	10

Для речових типів в таблиці наведені абсолютні величини мінімальних і максимальних значень.

Для написання стерпних на різні платформи програм не можна робити припущень про розмір типу int. Для його отримання необхідно користуватися операцією sizeof, результатом якої є розмір типу в байтах. Наприклад, для операційної системи, MS-DOS sizeof (int) дасть в результаті 2, а для Windows 9x або OS / 2 результатом буде 4.

У стандарті ANSI діапазони значень для основних типів не задаються, визначаються тільки співвідношення між їх розмірами, наприклад:

```
sizeof (float) <= sizeof (double) <= sizeof (long double)
sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long)
```

Різні види цілих і речових типів, що розрізняються діапазоном і точністю представлення даних, введені для того, щоб дати програмісту можливість найбільш ефективно використовувати можливості конкретної апаратури, оскільки від вибору типу залежить швидкість обчислень і обсяг пам'яті. Але оптимізована для комп'ютерів будь-якого одного типу програма може стати не переноситься на інші платформи, тому в загальному випадку слід уникати залежностей від конкретних характеристик типів даних.

Лекція 3. Базові конструкції структурного програмування.

В теорії програмування доведено, що програму для вирішення завдання будь-якої складності можна скласти тільки з трьох структур, що називаються слідуванням, розгалуженням і циклом. Цей результат встановлений Бойм і Якопіні ще в 1966 році шляхом докази того, що будь-яку програму можна перетворити в еквівалентну, що складається тільки з цих структур і їх комбінацій.

Слідування, розгалуження і цикл називають базовими конструкціями структурного програмування. Слідуванням називається конструкція, що представляє собою послідовне виконання двох або більше операторів (простих або складених). Розгалуження задає виконання або одного, або іншого оператора в залежності від виконання певної умови. Цикл задає багаторазове виконання оператора (рис. 3.1). Особливістю базових конструкцій є те, що будь-яка з них має тільки один вхід і один вихід, тому конструкції можуть вкладатися один в одного довільним чином, наприклад, цикл може містити проходження з двох розгалужень, кожне з яких включає вкладені цикли (рис. 3.2).

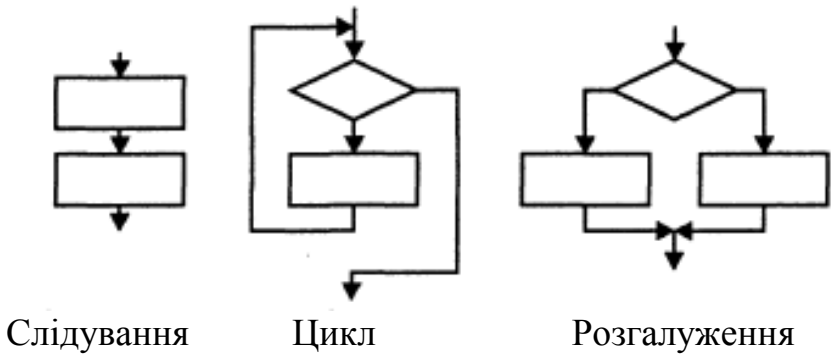


Рис. 3.1. Базові конструкції структурного програмування

Метою використання базових конструкцій є отримання програми простої структури. Таку програму легко читати (а програми частіше доводиться читати, ніж писати), налагоджувати і при необхідності вносити в неї зміни. Структурне програмування часто називали «програмуванням без goto», і в цьому є велика частка правди часте використання операторів передачі управління в довільні точки програми ускладнює простежування логіки її роботи. З іншого боку, ніякі принципи не можна зводити в абсолют, і є ситуації, в яких використання goto виправдано і призводить, навпаки, до спрощення структури програми.

У більшості мов високого рівня існує кілька реалізацій базових конструкцій; в C ++ є три види циклів і два види розгалужень (па два і на будь-яку кількість напрямків). Вони введені для зручності програмування, і в кожному випадку треба вибирати найбільш відповідні засоби. Головне, про що потрібно пам'ятати навіть при написанні найпростіших програм, - що вони повинні

складатися з чіткій послідовності блоків строго певної конфігурації. «Хто ясно мислить, той ясно викладає» - практика давно показала, що програми в стилі «потік свідомості» нежиттєздатні, не кажучи про те, що вони просто непривабливі.

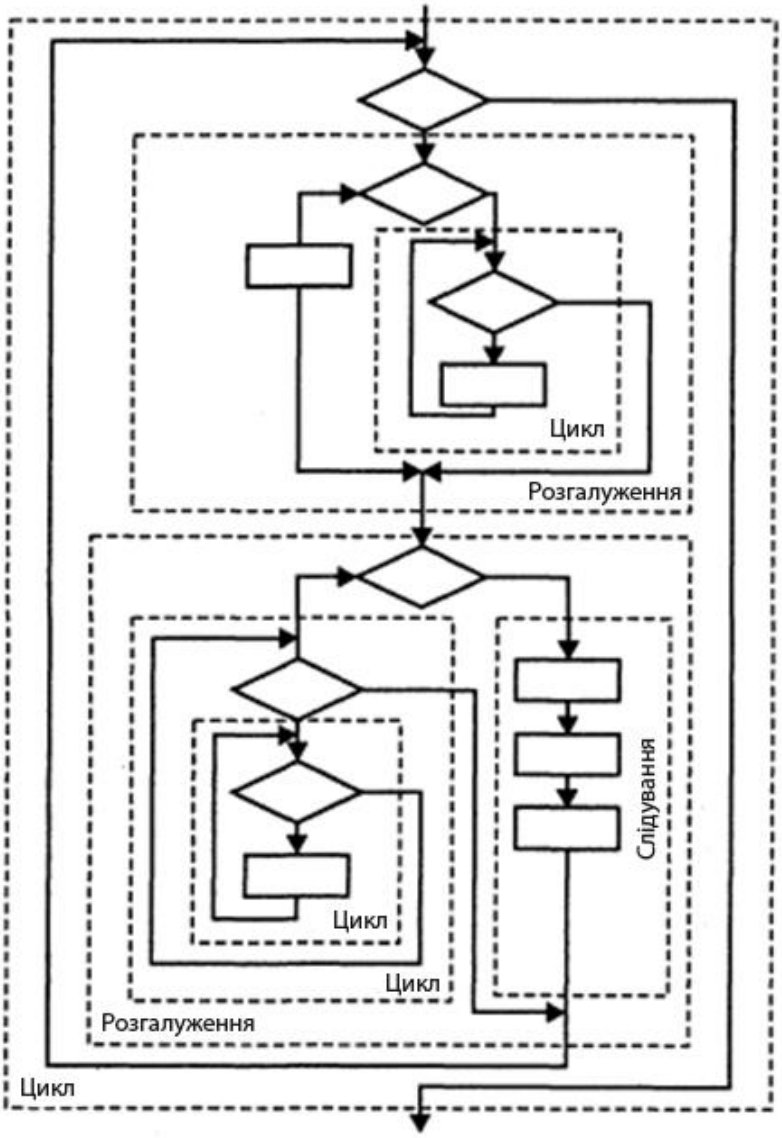


Рис. 3.2. Вкладення базових конструкцій

Розглянемо оператори мови, що реалізують базові конструкції структурного програмування.

3.1. Оператор «вираз»

Будь-яке вираження, що завершується крапкою з коми, розглядається як оператор, виконання якого полягає в обчисленні виразу. Окремим випадком виразу є порожній оператор; (Він використовується, коли але синтаксису оператор потрібно, а за змістом - ні). Приклади:

```
i ++;              // виконується операція інкремента
a * = b + c;       // виконується множення з присвоєнням
```

```
fun (i, k):          // виконується виклик функції
```

3.2. Оператори розгалуження

Умовний оператор if.

Умовний оператор if використовується для розгалуження процесу обчислень на два напрямки. Структурна схема оператора наведена на рис. 3.3. Формат оператора:

```
if (вираз) оператор_1, [else оператор_2;]
```

Спочатку обчислюється вираз, яке може мати арифметичний тип або тип покажчика. Якщо вона не дорівнює нулю (має значення true), виконується перший оператор, інакше - другий. Після цього управління передається па оператор, наступний за умовним.

Одна з гілок може бути відсутнім, логічніше опускати другу гілку разом з ключовим словом else. Якщо в будь-якої гілки потрібно виконати декілька операторів, їх необхідно укласти в блок, інакше компілятор не зможе зрозуміти, де закінчується розгалуження. Блок може містити будь-які оператори, в тому числі опису та інші умовні оператори (але не може складатися з одних описів). Необхідно враховувати, що змінна, описана в блоці, поза блоком не існує.

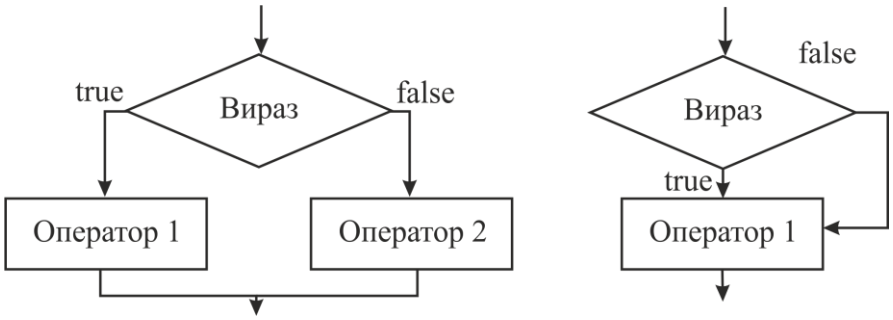


Рис. 3.3. Структурна схема умовного оператора

Приклад №3.1

```
if (a <0) b = 1:                                                    // (1)
if (a <b && (a> d || a == 0)) b ++: else {b * = a: a = 0;}          // (2)
if (a <b) {if (a <c) m = a: el se m = з:} else {if (b <c) m = b: else m = c;}  // (3)
if (a ++) b ++:                                                     // (4)
if (b> a) max = b: else max = a;                                    // (5)
```

У прикладі №3.1 (1) відсутня гілка else. Подібна конструкція називається «пропуск оператора», оскільки присвоювання або виконується, або пропускається в залежності від виконання умови.

Якщо потрібно перевірити кілька умов, їх об'єднують знаками логічних операцій. Наприклад, вираз в прикладі №3.1 (2) було це слово в тому випадку,

якщо виконається одночасно умова a <b і одна з умов в дужках. Якщо опустити внутрішні дужки, буде виконано спочатку логічне І, а потім - АБО.

Оператор в прикладі №3.1 (3) обчислює найменше значення з трьох змінних. Фігурні дужки в даному випадку не є обов'язковими, так як компілятор відносить частину else до найближчого if.

Приклад №3.1 (4) нагадує про те, що хоча в якості виразів в операторі if найчастіше використовуються операції відносини, це не обов'язково.

Конструкції, подібні оператору в прикладі №3.1 (5), простіше і наочніше записувати у вигляді умовної операції (в даному випадку: max = (b> a)? b: a;).

Приклад № 3.2.

```
//Робиться постріл по мішені. Визначити кількість очок.

#include <i ostream. h>

int main () {
        float x. y: int kol:
        cout << "Введіть координат пострілу \ n":
        cin >> x >> y:
        if (X * x + y * y <1)  kol = 2;
        else if (x * x + y * y <4)     kol = 1;
        else kol = 0:
        cout «" \ п Точки: "« kol:
return 0:
}
```

Тип змінних вибирається виходячи з їх призначення. Координати пострілу не можна уявити цілими величинами, так як це призведе до втрати точності результату, а лічильник очок не має сенсу описувати як речовинний. Навіть таку просту програму можна ще спростити за допомогою проміжної змінної і записи умови у вигляді двох послідовних, а не вкладених, операторів if (зверніть увагу, що в першому варіанті значення змінної kol присвоюється одно один раз, а в другому-від одного до трьох разів в залежності від виконання умов):

```
#include <iostream.h>
        int main () {
        float x, y, temp: int kol;
        cout << "Введіть координати пострілу \ n";
        cin >> x >> y;
        temp = x * x + y * y;
        kol = 0;
        if (temp <4) kol = 1;
```

```
        if (temp <1) kol = 2;
        cout << "\ n Точки:" << kol;
        return 0;
}
```

Якщо яка-небудь змінна використовується тільки всередині умовного оператора, рекомендується оголосити її всередині дужок, наприклад:

```
if (int i = fun (t)) a - = i; else a + = i;
```

Оголошення змінної в той момент, коли вона потрібна, тобто коли їй необхідно присвоїти значення, є ознаками хорошого стилю і дозволяє уникнути випадкового використання змінної до її ініціалізації. Оголошувати всередині оператора if можна тільки одну змінну. Область її видимості починається в точці оголошення і включає обидві гілки оператора.

Оператор switch.

Оператор switch (перемикач) призначений для розгалуження процесу обчислень на кілька напрямків. Структурна схема оператора наведена на рис. 3.4. Формат оператора:

```
switch (вираз) {
case константний_ вираз_1: [список_операторів_ 1]
case константний_ вираз_2: [список_операторів _2]

case константний_вираз_n: [список_операторів_n]
[default: оператори]
}
```

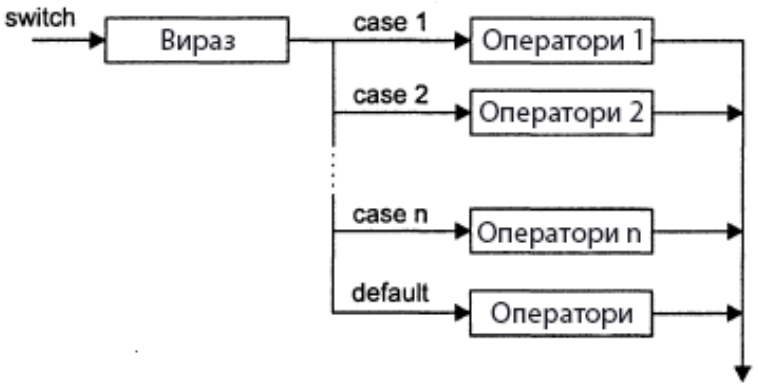


Рис. 3.4. Структурна схема оператора switch

Виконання оператора починається з обчислення виразу (воно повинно бути цілочисельним), а потім управління передається першому оператору зі списку, позначеного константним виразом, значення якого співпало з

обчисленим. Після цього, якщо вихід з перемикача явно не вказано, послідовно виконуються всі інші гілки.

Вихід з перемикача зазвичай виконується за допомогою операторів break або return. Оператор break виконує вихід з самого внутрішнього з осяжний його операторів switch, for, while і do. Оператор return виконує вихід з функції, в тілі якої він записаний.

Всі вирази зі сталими повинні мати різні значення, але бути одного і того ж цілочисленного типу. Кілька міток можуть слідувати поспіль. Якщо збіги не відбулося, виконуються оператори, розташовані після слова default (а при його відсутності управління передається наступному за switch оператору).

Приклад №3.3

```
// (Програма реалізує найпростіший калькулятор на 4 дії):

#include <iostream. h>
    Int main () {
    Int a, b, res;
    char op;
    cout << "\ nВведіте 1й операнд:"; cin >> a;
    cout << "\ nВведіте знак операції:"; cin »op;
     cout << "\ nВведіте 2й операнд:"; cin »b;
    bool f = true;
    switch (op) {
        case '+': res = a + b; break;
        case '-': res = a - b; break;
        case '*': res = a * b; break;
        case '/': res = a / b; break;
        default: cout «" \ n Невідома операція ": f = false;
}
if (f) cout «" \ nРезультат: "« res;
return 0;
```

3.3. Оператори циклу

Оператори циклу використовуються для організації багаторазово повторюваних обчислень. Будь-який цикл складається з тіла циклу, тобто тих операторів, які виконуються кілька разів, початкових установок, зміни параметру циклу і перевірки умови продовження виконання циклу (рис. 3.5).

Один прохід циклу називається ітерацією. Перевірка умови виконується на кожній ітерації або до тіла циклу (тоді говорять про цикл з передумовою), або після тіла циклу (цикл з умовою поста). Різниця між ними полягає в тому, що тіло циклу з умовою поста завжди виконується хоча б один раз, після чого перевіряється, чи треба його виконувати ще раз. Перевірка необхідності

виконання циклу з передумовою робиться до тіла циклу, тому можливо, що він не виконається жодного разу.

Рис. 3.5. Структурні схеми операторів циклу: а - цикл з передумовою; б - цикл з умовою постумовою

Змінні, що змінюються в тілі циклу і використовуються при перевірці умови продовження, називаються параметрами циклу. Цілочисельні параметри циклу, що змінюються з постійним кроком на кожній ітерації, називаються лічильниками циклу.

Початкові установки можуть явно не бути присутнім в програмі, їх зміст полягає в тому, щоб до входу в цикл задати значення змінним, які в ньому використовуються.

Цикл завершується, якщо умова його продовження не виконується. Можливо примусове завершення як поточної ітерації, так і циклу в цілому. Для цього служать оператори break, continue, return і goto. Передавати керування ззовні всередину циклу не рекомендується.

Для зручності, а не за потребою, в C ++ є три різних оператора циклу - while, do while і for.

Цикл з передумовою (while).

Цикл з передумовою реалізує структурну схему, наведену на рис. 3.5а, і має вигляд:

```
while (вираз) оператор
```

Вираз визначає умову повторення тіла циклу, представленого простим або складеним оператором. Виконання оператора починається з обчислення виразу.

Якщо воно істинне (не дорівнює false), виконується оператор циклу. Якщо при першій перевірці вираз дорівнює fa1se, цикл не виконається жодного разу. Тип виразу повинен бути арифметичним або приводиться до нього. Вираз обчислюється перед кожною ітерацією циклу.

Приклад №3.4

//(Програма друкує таблицю значень функції y = x^2 + 1 у введеному діапазоні)

```
#include <stdioh>
int main () {
float Xn, Xk, Dx;
printf ( "Введіть діапазон і крок з трансформаційних змін аргументу:");
scanf ( "% f% f% f". & Xn. & Xk. & Dx);
printf ( "|     X        |        Y        | \ N ");       // шапка таблиці
float X = Xn;                                               // установка пара метра циклу
while (X <= Xk) {                                           // перевірка умови продовження
printf ( "|% 5.2f |% 5.2f | \ n". X. X * X + 1): // тіло циклу
X + = D x:                                                  // модифікація параметра
}
return 0:
}
```

Приклад №3.5.

// (Програма знаходить все подільники цілого позитивного числа)

```
#include <iostream.h>
int main () {
int num:
cout << "\ nВведіте число:"; cin >> num;
int half = num / 2;                                         // половина числа
int di v = 2;                                               // кандидат на дільник
while (div <= half) {

if (! (num% di v)) cout << div << "\ n":
div ++;
}
return 0;
}
```

Поширений прийом програмування - організація нескінченного циклу з заголовком while (true) або while (1) і примусовим виходом з тіла циклу з виконання певної умови.

В круглих дужках після ключового слова while можна вводити опис змінної. Областю її дії є цикл:

```
while (int x = 0) {/ * область дії x * /}
```

Цикл з умовою поста (do while).

Цикл з умовою поста реалізує структурну схему, наведену на рис. 3.5б, і має вигляд:

```
do оператор while вираз;
```

Спочатку виконується простий або складений оператор, що становить тіло циклу, а потім обчислюється вираз. Якщо воно істинне (не дорівнює false), тіло циклу виконується ще раз. Цикл завершується, коли вираз стане рівним false або в тілі циклу буде виконано будь-якої оператор передачі управління. Тип виразу повинен бути арифметичним або приводиться до нього.

Приклад №3.6

```
// (Програма здійснює перевірку введення)

#include <iostream. h>
int main () {
        char answer:
        do {
                cout «" \ nКупі слоника! "; cin >> answer:
        } While (answer! = 'y');
        return 0;
}
```

Цикл з параметром (for).

Цикл з параметром має такий вигляд:

```
for (ініціалізація: вираз; модифікації) оператор;
```

Ініціалізація використовується для оголошення і присвоєння початкових значень величинам, використовуваним в циклі. У цій частині можна записати кілька операторів, розділених комою (операцією «послідовне виконання»), наприклад, так:

```
for (int i = 0. j = 2: _...
```

```
int k, m;
for (k = 1, m = 0; ... _
```

Областю дії змінних, оголошених в частині ініціалізації циклу, є цикл. Ініціалізація виконується один раз на початку виконання циклу.

Вираз визначає умову виконання циклу: якщо його результат, приведений до типу bool, дорівнює true, цикл виконується. Цикл з параметром реалізований як цикл з передумовою.

Модифікації виконуються після кожної ітерації циклу і служать зазвичай для зміни параметрів циклу. У частині модифікацій можна записати кілька операторів через кому. Простий або складовою оператор являє собою тіло циклу. Будь-яка з частин оператора for може бути опущена (але крапки з комою треба залишити на своїх місцях!).

Приклад (оператор, який обчислює суму чисел від 1 до 100):

```
for (int i = 1, s = 0; 1 <= 100; I ++) s + = i;
```

Приклад (програма друкує таблицю значень функції $y = x^2 + 1$ у введеному діапазоні):

```
#include <stdioh>
int main () {
        float Xn, Xk, Dx, X;
        printf ( "Введіть діапазон і крок з трансформаційних змін аргументу:");
        scanf ( "% f% f% f", & Xn, & Xk, & D x);
        printf ( "|     X        |        Y        | \ N ");
        for (X = Xn; X <= Xk; X + = D x);
        printf ( "|% 5 .2f |% 5.2f | \ n". X. X * X + 1);
        return 0;
}
```

Будь-який цикл whi1e може бути приведений до еквівалентного йому циклу for і навпаки за наступною схемою:

```
for (b1; b2; b3) оператор        b1;
                                 while (b2) {
                                 оператор: b3;}
```

Помилки, що спостерігаються при програмуванні циклів - використання в тілі циклу неініціалізованих змінних і невірний запис умови ви ходу з циклу.

Щоб уникнути помилок, рекомендується:

- перевірити, чи всім змінним, яке трапляється в правій частині операторів присвоювання в тілі циклу, присвоєні до цього початкові значення (а також чи можливо виконання інших операторів);
- перевірити, чи змінюється в циклі хоча б одна змінна, що входить в умову виходу з циклу;
- передбачити аварійний вихід з циклу по досягненню деякої кількості ітерацій;
- і, звичайно, не забувати про те, що якщо в тілі циклу потрібно виконати більше одного оператора, потрібно укладати їх в фігурні дужки.

Оператори циклу взаємозамінні, по можна навести деякі рекомендації по вибору найкращого в кожному конкретному випадку.

Оператор do while зазвичай використовують, коли цикл потрібно обов'язково виконати хоча б раз (наприклад, якщо в циклі проводиться введення даних).

Оператором while зручніше користуватися у випадках, коли число ітерацій заздалегідь не відомо, очевидних параметрів циклу немає чи внесення змін до параметрів зручніше записувати не в кінці тіла циклу.

Оператор for краще в більшості інших випадків (однозначно - для організації циклів з лічильниками).

Лекція 4. Базові структури. Область видимості

4.1. Область видимості

Кожне ім'я (ідентифікатор) у C++-коді (подібно до будь-якої мови програмування) повинно виражати унікальну фізичну чи логічну сутність (змінну, об'єкт, функцію, тип або шаблон). Це не є вимога унікальності імені: у коді, за умови існування деякого контексту, який за означенням дає змогу однозначного трактування значення, один і той самий ідентифікатор, може використовуватись для представлення різних сутностей. Таким контекстом у мові програмування є область видимості (ОВ) - ділянка коду, у якій ідентифікатор можна використовувати безпосередньо. Іноді поняття ототожнюють з областю дії але для ідентифікаторів.

Мова C++ підтримує три види ОВ: локальна область видимості (ЛОВ), глобальна область видимості просторів імен (ГОВПІ), область видимості класу (ОВК).

У випадку використання терміну контексту види ОВ мають назви: локальний контекст, глобальний контекст та контекст класу.

Локальною областю називають ділянку вихідного коду, яка визначає функцію, або програмний блок. Така область є у будь-якій функції. Вона або визначає функцію (її тіло), або знаходиться усередині тіла і визначають окрему локальну область.

Областю видимості простору імен називають ділянку вихідного коду, яка, на відміну від випадку локальної області, не міститься усередині оголошення або визначення функції чи класу. Поряд з існуванням глобальної області імен, розробнику надаються засоби для створення користувацький просторів імен, ідентифікатори яких також є глобальними. Такі простори визначають область, яку називають областю видимості простору імен (ОВПІ) Але користувацькі простори імен завжди розглядаються в контексті вкладення у глобальну область. Кожен такий простір є окремою, вкладеною у ГО, областю видимості.

Об'єкти, функції, типи та шаблони повинні визначатись у глобальній області видимості. Незважаючи на те, що, наприклад, користувацький тип може бути оголошений у тілі функції, рекомендації по програмуванню явно вказують розробнику на те, що ідентифікатор будь-якого типу повинен бути глобальним. Це зумовлено тим, що за межами локальної області видимості ім'я типу стає недоступним незважаючи на те, що інформація про тип.

Областю видимості класу називають ділянку коду, яка визначає клас. Будь-який ідентифікатор визначений класі є видимим у цілому класі незалежно від місця оголошення. Очевидно, що кожне визначення класу являє визначаю окрему область видимості класу.

У залежності від області видимості ім'я може виражати різні сутності. У наступному фрагменті програми ідентифікатор `s1` відноситься до чотирьох різних сутностей:

Приклад № 4.1.

```
#include <iostream>
// Визначаємо псевдоім'я - тип функція
typedef int (ALG) (char* [], char* []);
// Порівнюємо s1 і s2 за методом 1
int sort_method1 (char* s1[], char* s2[]) { . . . }
// Порівнюємо s1 і s2 за методом 2
int sort_method2 (char* s1[], char* s2[]) {. . .}
// Сортуємо масив рядків
bool sort (char* s1[], ALG algorithm = sort_method1)
{
return algorithm (s1, s1) == 0 ? true : false ;
}

// Визначаємо глобальну змінну s1
char *s1[10] = {"a", "light", "drizzle", "was", "falling",
"when", "they", "left", "the", "school"}; void main(void)
{
if ( sort(s1) )
for(unsigned i = 0; i < 10; ++i) std::cout << s1[i] << "\n \t";
return 0;
}
```

Визначення функцій sort_method1(), sort_method2(), sort(), main() а також оператора for є різними області видимості. Вони усі є відмінними від глобальної області і у кожній з цих них можна визначити свою змінну з іменем s1. Тут треба зазначити, що визначати саме таку сутність як змінна є необов'язковим. Ім'я s1 у іншій ОВ може виражати зовсім іншу сутність. Наприклад, у тілі функції sort_method2() ідентифікатор s1 можна використати для визначення типу

```
// Порівнюємо s і s2 за методом 2
int sort_method2 (char* s[], char* s2[])
{
struct s1
{} a;
. . .
}
```

Введений ідентифікатор в межах ОВ і ділянки від точки оголошення до кінця області видимості (включаючи вкладені області) може використовуватись без жодних обмежень. Так, ім'я s1 параметра функції sort_method1() можна використовувати області видимості, тобто до кінця визначення функції. А ім'я

глобального масиву s1 - від точки його оголошення до кінця вихідного файлу, включаючи вкладені області, такі, як тіло функції main().

Загалом ідентифікатор всередині однієї області видимості повинен однозначно виражати лише одну сутність. Якщо в Прикл. № 4.1 у глобальній області видимості, наприклад додати такий рядок

```
void s1 (void)  { }
```

то компілятор видасть повідомлення про помилку, оскільки в одній області видимості (ГОВПІ) була спроба дві сутності (функцію та масив) виразити одним ідентифікатором s1.

Подібно до більшості мов з компільованим кодом у C++ ідентифікатор повинен бути оголошений до моменту його першого використання. Процедура зіставляння імені, яке використовується у виразі, з його оголошенням називається дозволом дії і в окремих випадках виражається явно оператором operator::.

Дозвіл дії для імені залежить від способу його використання та від його області видимості За допомогою цієї процедури в ідентифікатор вкладається конкретний зміст. Відбувається так звана конкретизація ідентифікатора.

У контексті процедури конкретизації треба підкреслити таке: область видимості та і дозвіл дії - це поняття, які стосуються виключно ідентифікаторів і лише на періоді компіляції. Вони мають стосунок лише до окремих ділянок вихідного коду. Компілятор інтерпретує текст програми згідно з правилами областей видимості і правилам дозволу дії.

Як вже визначалось ЛОВ є частиною коду, яка визначається тілом функції. Це означає, що з одного боку усі функції мають власні локальні області видимості, а з іншого - саме тіло функції або будь-який програмний блок (або складений оператор), які у ньому визначені, є ЛОВ. Області, які визначаються у тілі функції називаються вкладеними ЛОВ. У свою чергу, вони можуть містити інші вкладені ЛОВ.

Будь-яка ЛОВ повинна завжди розглядатись вкладеною. Наприклад тіло функції у контексті глобальної ОВ є вкладеною ЛОВ, оскільки має місце входження у ГОВПІ. Тому у випадках вкладень областей видимостей вводять поняття рівнів17. Проілюструємо це на прикладі. Наступний код містить три рівні (рахунок ведеться від 0) вкладення областей видимості:

Приклад № 4.2.

```
#include <iostream> using namespace std;
// Глобальна область видимості: рівень # 0
const int fnd = -1; // глобальний контекст
int search (int* vec, unsigned int s, int val)
{
// Локальна область видимості: рівень # 1
```

```
if ( (!vec) || (s == 0) ) return fnd;
int low = 0,
high = (int)s - 1; while (low <= high)
{
// Локальна область видимості: рівень # 2
// видимими є ідентифікатори:
// локальні vec, s, val та глобальний fnd
int mid = (low + high) / 2;
// до видимих додано локальний ідентифікатор mid
if (val < vec [mid])
high = mid - 1;
else if (val > vec [mid])
low = mid + 1; else
return mid;
}
// локальна область видимості: рівень # 1
// вихід за межі області рівня # 2
return fnd;
}
// глобальна область видимості: рівень # 0
// вихід за межі області рівня # 1
void main(void)
{
// Локальна область видимості: рівень # 1
int val[] = {1, 2, 3, 4, 5, 6, 7};
// видимими є
// локальний ідентифікатори val та глобальний fnd
{
// Локальна область видимості: рівень # 2
int m = 4,
in;
// до видимих додано локальні ідентифікатори in, in
if ((in = search(val, 7, m)) >= 0)
cout << "Value " << m <<" has index - " << in << " in massive" << endl;
else
cout << "Value " << m <<" isn't exist in massive" << endl;
}
// локальна область видимості: рівень # 1
// вихід за межі області рівня # 2
// видимими є
// локальний ідентифікатори val та глобальний fnd
```

```
// m++; дія невалідна - ідентифікатор m є невидимим
val[0]++; // дія валідна - ідентифікатор val є видимим
}
// глобальна область видимості: рівень # 0
// вихід за межі області рівня # 1
```

У наведеному коді на нульовому рівні визначена глобальна область, і локальні області 1-го та другого рівнів. Одні із ЛО є вкладеними у ГО, а інші - у локальні області нижчого рівня. Окрім локальних областей, ГО містить ідентифікатор *fnd*, який виражає константу цілого типу. Перша локальна область видимості (рівень #1) - тіло функції *search*(). У ній через оголошення параметрів функції визначено три ідентифікатори *vec*, *s*, *val*. Додатково у тілі функції через створення локальних змінних визначено ще два ідентифікатори *low*, *high*. Більше цього, у тілі функції цикл while визначає ще одну локальну область (рівень #2), яка є вкладеною у область рівня #1. У цій області створення змінної виражає ще один ідентифікатор mid. Параметри vec, s, val і змінні low, high є видими у вкладеній області рівня #2.

Імена параметрів функції vec, s і val належать до першого локального контексту (контекст рівня #1)) - тіла функції. У цій ЛОВ, тобто в межах однієї області видимості, використовувати ті ж імена для вираження інших або нових сутностей забороняється. Наприклад:

```
int search (int* vec, unsigned int s, int val)
{
// Локальна область видимості: рівень # 1
char vec; // !!!помилка - інша сутність (вона також нова)
int vec; // !!! помилка - нова сутність
...
```

У той же час у іншій ЛО, яка не зв'язана з попередньою відношення вкладення це не забороняється. Наприклад, у визначенні функції search() ідентифікатор val виражає змінну, у тілі функції main() - локальний статичний масив цілих чисел.

Імена параметрів є видимим усюди в тілі функції search() зокрема всередині вкладеної області видимості рівня #2 (цикл while). Але вони є невидимими за межами функції search(). Дозвіл дії імені в локальній області видимості тіла функції search() відбувається так. На початку аналізується та область, де використовувався ідентифікатор. Якщо оголошення ідентифікатора знайдено у цій області, то використання імені є дозволеним. Якщо ні, то аналізується зовнішня область видимості, яка включає поточну. Цей процес продовжується доти, доки оголошення ідентифікатора не буде знайдено або не буде досягнута глобальний контекст. Якщо і у цій області оголошення є

відсутнім, то ідентифікатор вважається помилковим і дозвіл дії з ним не надається.

Описаний порядок перегляду областей видимості дає можливість реалізувати технологію приховування ідентифікаторів. Саме через порядку перегляду областей видимості в процесі дозволу дії оголошення ідентифікатору у зовнішній області може бути приховано (перекрито) оголошенням такого імені у вкладеній області. Наприклад, якщо б у наведеному прикладі ідентифікатор low , незалежно від сутності був оголошений у ГО (тобто перед визначенням функції search()), то використання імені low у локальній області видимості (наприклад у циклі while) стосується лише локального оголошення і приховує глобальне оголошення. Це ілюструє такий код:

```
// Глобальна область видимості: рівень # 0
...
// Глобальне визначення ідентифікатора low
struct low
{ } a;
int search (int* vec, unsigned int s, int val)
{
// Локальна область видимості: рівень # 1
...
// Локальна визначення ідентифікатора low
int low = 0,high = (int)s - 1;
// Використання локального ідентифікатора low
while (low <= high)
{
...
        }
```

Деякі інструкції мови *C++* у керуючій частині можуть оголошувати ідентифікатори, які виражають змінні. Наприклад, модифікуємо функцію *search*() таким чином, щоб її тіло містило циклічний оператор for. У списку визначення цього оператора, зокрема у частині ініціалізації, оголосимо ідентифікатор *i*, який виражатиме змінну. Ця змінна буде використовуватись лічильником ітерацій оператора

```
...
for (int i = 0; i < size; i++)
{
// Локальна ОВ оператора for
// Ідентифікатор i видимий лише межах цієї ЛОВ
// Він може виражати сутність - змінну
```

```
// У тілі ЛОВ створюються нові сутності
// Вони можуть бути різними,
// Їх ОВ - це ЛОВ оператора for
struct B                    // Ідентифікатор нової сутності - структури
{
int i, error;
} b;                        // Ідентифікатор нової сутності - змінної
if ( vec[i] == val )
{
b.i = i;
b.error = 0;
return b;
}
}
// Помилка використання ідентифікаторів i та b.
// За межами ЛОВ вони є невидимими.
        if (i! = size )     // елемент не знайдено
{
b.i = i;
b.error = fnd;
return b;
}
...
```

Тіло оператора for визначає ЛОВ. А тому, ідентифікатори, які оголошені або у списку визначення (тоді вони можуть виражати лише таку сутність як змінна), або у тілі оператора (тоді вони можуть виражати будь-які сутності) є видимим у ЛОВ, яка визначається тілом оператором і в усіх вкладених у тіло ЛОВ. За межами тіла оператора (або за межами вкладених ЛОВ) такі ідентифікатори є невидимими. Тому їх використання є помилкою. Приведемо схему розглядання компілятором конструкцію оператора for:

```
// Розгляд компілятором
{                           // Невидимий блок (ЛОВ оператора for) - початок
int i = 0;                  // Ідентифікатор нової локальної сутності
struct B                    // Ідентифікатор нової локальної сутності
{
int i, error;
} b;                        // Ідентифікатор нової локальної сутності
for (; i < size; i++)
{                           // Тіло оператора for - початок
```

```
...
}                          // Тіло оператора for - закінчення
}                          // Невидимий блок - закінчення
// Помилка використання ідентифікаторів i та b.
// За межами ЛОВ вони є невидимими.
if (i! = size )            // елемент не знайдено
...
```

Зважаючи на це розробник не може використовувати ідентифікатори i, B та b за межами тіла оператора. Якщо використання цих ідентифікаторів є обов'язковим, то для розглядуваного випадку можливі два шляхи вирішення цієї проблеми:

- перший з них полягає у явному записі наведеного розгляду компілятора і занесення у ЛОВ оператора if разом із його тіла.
- другий шлях полягає у неоголошені ЛОВ. Тоді визначення сутностей повинно бути за межами оператора for, наприклад

```
int i = 0; // Ідентифікатор нової сутності
struct B // Ідентифікатор нової сутності
{
int i, error;
} b; // Ідентифікатор нової сутності
for (; i < size; i++)
{ // Тіло оператора for - початок
...
} // Тіло оператора for - закінчення
// Помилки немає.
// Допустиме використання ідентифікаторів i та b.
// За межами ЛОВ оператора for вони є видимими.
if (i! = size ) // елемент не знайдено
        ...
```

Оскільки оператор for визначає окрему ЛОВ, то ідентифікатори сутностей цієї ЛОВ можуть використовуватись для вираження сутностей у інших у тому числі вкладених ЛОВ. Наприклад

Приклад № 4.3.

```
#include <iostream>
// Глобальна область видимості: рівень # 0
using namespace std;
void executing ( int val )
{
```

```
// Локальна область видимості: рівень # 1
val++;
cout << "Fuctional Parameters: "        << val <<" "<< endl; for (int val = 1; val <=
2; val++)
// Локальна область видимості: рівень # 2
cout << "First for: " << val <<" " << endl; for (int val = 3; val <= 4; val++)
// Локальна область видимості: рівень # 2
cout << "Second for: " << val <<" " << endl;
{
// Локальна область видимості: рівень # 2
int val = 5;
cout << "Local variable: " << val <<" "<< endl;
}
cout << "Fuctional Parameters: "        << val <<" "<< endl;
}
void main(void)
{
// Локальна область видимості: рівень # 1
int val = -2;
executing( val );
cout << "Main's variable: "        << val <<" "<< endl;
}
```

Існування цій програмі багатьох сутностей, для яких використовується один ідентифікатор val є можливим завдяки тому, що ці сутності створюються у різних ЛОВ.

Описана ситуація може бути використана для багатьох програмних конструкцій. Але, оператори, які визначаються ці конструкції, повинні бути такими, які можуть розглядатись у контексті ЛОВ. Зокрема це оператори if, switch, або while. Наприклад:

```
...
// Зовнішня область видимості: рівень # 0
// Створення локальної сутності у ЛОВ, утвореної оператором if
if ( int y = (double)rand() )
// Локальна область видимості: рівень # 1
y--;    // Ідентифікатор є видимим
else if ( y > 5 )        // Ідентифікатор є видимим
// Попередня локальна область видимості: рівень # 1
{
// Локальна вкладена у №1 область видимості: рівень # 2
y++;   // Ідентифікатор є видимим
```

```
int d = 0;      // Створення локальної сутності у ЛОВ
}
// Попередня локальна область видимості: рівень # 1
else
{
// Локальна вкладена у №1 область видимості: рівень # 2
y += 2;         // Ідентифікатор є видимим
d++;   // Помилка - ідентифікатор є не видимим
}
// Зовнішня область видимості: рівень # 0
y = 0;  // Помилка - ідентифікатор є не видимим
...
```

Визначені у конструкції оператора if ідентифікатори є видимими лише всередині ЛОВ, яка, у свою чергу, визначається тілом оператора. За межами цієї ЛОВ ідентифікатори стають невидимими (у прикладі це спроба доступу до змінної y, яка спричинить помилку у рядку y = 0). До цієї ЛОВ належать усі вітки оператора. Кожна з віток може утворювати власну ЛОВ, яка буде вкладеною у область оператора (у прикладі це ЛОВ віток elseif та else). Створення будь-яких локальних сутностей у цих ЛОВ призведе до того, що їх ідентифікатора не будуть видимими за межами областей (у прикладі це помилка у рядку d++).

4.2. Простори імен

По замовчуванні будь-який об'єкт, функція, тип (чи шаблон) оголошений у глобальній області видимості вводить глобальну сутність. Однією із слабких сторін ANSI C було наявність у програми лише одного простору імен. Це означає, що розробник був змушений уникати дублювання імен ідентифікаторів, які мають ОВ або область дії (ОД), що перетинаються.

Задача уникнення перетинання ОВ або ОД ще сильніше ускладнювалась ситуаціями, коли інтенсивно використовувались бібліотеки сторонніх виробників. У них імена, визначені в одній бібліотеці, конфліктували з іменами, визначеними в іншій. Усі проблеми вирішило введення поняття простору імен (namespace).

ОД це поняття аналогічне ОВ для випадку, коли ідентифікатор виражає таку фізичну сутність як змінну. У багатьох книгах простори імен розглядають стосовно змінних, а для них базовою характеристикою виступає ОД. Тому для пояснення цієї технології найчастіше використовується саме поняття області дії. Але ідентифікатор може виражати іншу сутність ніж змінна. Тому у технології просторів імен повинно використовувати поняття ОВ. Усі оголошення у просторі імен є глобальними, а тому їх ОД завжди є глобальний простір імен. Відповідно використовувати поняття області дії у контексті просторів імен є не зовсім коректно.

Простором імен (ПІ) або іменованою областю називатимемо технологію, яка використовується для локалізації імен ідентифікаторів з метою уникнення колізій (name collisions) існування однойменних ідентифікаторів. Фактично ПІ визначає користувацьку ОВ ідентифікаторів. Відповідно кожен ПІ стає окремою ОВ.

Загальний синтаксис визначення ПІ має вид

```
namespace NameSpace;
{
// декларації
}
```

Усі сутності, оголошені всередині ПІ, називаються членами простору імен. Подібно до випадку глобального простору будь-який ідентифікатор, який використовується для визначення членів ПІ, повинен бути унікальним в межах самого простору.

ПІ може містити вкладені ПІ, оголошення чи визначення функцій, об'єктів (змінних) та типів (шаблонів). Основним призначенням ПІ є локалізація дії ідентифікаторів, у першу чергу змінних (об'єктів). Це дає можливість використовувати одне й те саме ім'я можна використовувати в різних безконфліктних контекстах. Важливо розуміти, що оголошення змінної всередині простору імен виокремлює ділянку від точки оголошення і до кінця оголошення самого простору, у якій ідентифікатор є видимим без додаткових специфікаторів доступу. За межами цієї ділянки ідентифікатор або недоступний або доступний при використання різноманітних специфікаторів та технологій доступу.

Синтаксис простору імен передбачає оголошення ідентифікаторів глобальної області дії. Якщо ці ідентифікатори виражають змінні, то вони будуть ініціалізовані по замовчуванні подібно до змінних класу пам'яті extern. При потребі ініціалізацію можна здійснити у явному виді.

Сформулюємо основні правила, яких треба дотримуватись при використанні ПІ:

- ПІ оголошується лише глобально; не існує поняття локального ПІ;
- простори імен можуть бути вкладеними. Проте заборонено оголошувати ПІ всередині будь-якої іншої області видимості;
- ПІ може розширюватись як у даному файлі так і у інших ФЗ;
- можна задавати ім'я, яке визначатиме даний ПІ і через це ім'я викликати змінні цього простору, наприклад
- для задіяння конкретного ПІ можна використовувати директиву using;
- не можна створювати екземпляри ПІ;
- усі оголошення в просторі імен є відкритими. Стосовно вищевикладеного наведемо приклад

Приклад № 4.4.

```
namespace SP_1 // оголошення ПІ з іменем SP_1
{ // який складається зі
typedef char C; // псевдоімені
int i; // змінної i
int f(); // функції f()
union A // та типу A
{
int ui;
C uc;
};
}
namespace SP_2 // оголошення з іменем SP_2
{ // який складається зі
int i = 2; // змінної i
int f() // функції f()
{
return 2;
}
struct S // типу S
{
int si;
SP_1::C uc; // (використання псевдоімені з SP_1)
} s; // та об'єкту s
}
namespace SP_1 // розширення ПІ SP_1
{
int j; // додаванням змінної j
}
// реалізація функції f ПІ SP_1
int SP_1::f()
        {
return 1;
}
void main()
{
SP_1::C v; // використання псевдоімені з ПІ - SP_1
SP_1::j = 1; // доступ до змінної j з ПІ - SP_1
SP_1::A a; // доступ типу A з ПІ - SP_1
SP_2::s.si = 3; // використання об'єкта з ПІ - SP_2
using SP_1::i; // відкриття лише змінної i з ПІ - SP_1
```

```
i = SP_2::i-SP_1::j; // відкрита i з ПІ - SP_1
// i з ПІ - SP_2 та j з ПІ - SP_1
using namespace SP_1; //відкрття усього ПІ - SP_1
//доступ до f()з ПІ - SP_1 як до відкритої
// а до f() з ПІ - SP_2 як до закритої
return f()+SP_2::funct();
        }
```

Наведений приклад має одну особливість; в ньому використовуюся два простори імен SP_1 і SP_2. При сумісному використанні декількох ПІ простори не підміняють один одного, а просто додають у глобальний простір імена, які вони містять. У Прикл. № 4.2 глобальний ПІ складався з імен, які входять до складу трьох просторів SP_1, SP_2 і std.

Використання простору і змінних оголошених в ньому можливе за умови використання оператора дозволу дії (ОДД) operator::, наприклад SP_1::j= 1. З іншого боку простір можна відкрити за допомогою директиви, наприклад, using namespace A. Тоді усі змінні відкритого простору (в даному випадку A) є доступні без оператора ОДД.

У наведеній програмі функція f() з простору SP_1 оголошувалась і визначалась всередині самого простору. Проте всередині простору може бути лише попереднє оголошення функції — її прототип (наприклад, int f();). А повне визначення функції повинно знаходитись за межами простору (наприклад, int SP_1::f() {...}). Таке оголошення є типовим у випадку використання ФЗ. У цьому випадку, типово, сам простір (наприклад, B) оголошується у ФЗ, а повне визначення функцій простору (наприклад, B::f()) — у відповідному файлі коду. Це означає, що існує можливість розбити код простору імен на інтерфейсну частину і його реалізацію.

Наведемо приклад багатофайлового проекту із визначенням, реалізацією та використанням простору імен у різних файлах

Приклад № 4.5.

Файл - var.cpp

```
#include "var.h"
#include <iostream> void A::out();
{
std::cout << "Namespace A" << std::endl;
}
```

Файл - var.h

```
#ifndef VARH
```

```
#define VARH
namespace A
{
// int i; - недопустимі оголошення
void out();
}
#endif
```

Файл - run.cpp

```
#include <iostream>
#include "var.h"
void main ()
{
A::out();
}
```

Треба пам'ятати, що ідентифікатор функції за межами простору і незалежно від його відкритості є доступним лише через використання ОДД з іменем простору (наприклад, A::out()).

Це означає, що такі видозміни у Прикл. № 4.5

```
// реалізація функції out ПІ A
using namespace A;
void out()
{
std::cout << "Namespace A" << std::endl;
}
```

є неправильними.

Закривається простір шляхом відкривання нового простору. Але він ніколи не стає повністю закритим. Відповідно, новий простір не стає повністю відкритим. Прямого доступу немає лише до змінних, які були перекриті у новому просторі. Наприклад

```
namespace A
{
int i,z;
}

namespace B
{
int i, j;
```

```
}
...
using namespace A;
i = 1; z = 0; // i та z з ПІ - A
using namespace B;
//A::i--; i++27; j = 1; z++; // неправильний доступ
A::i--; B::i++; j = 1; z++; // правильний доступ
```

Наочним поясненням процесів закривання і відкривання просторів імен є діаграма, яка наведена на Рис. 4.1.

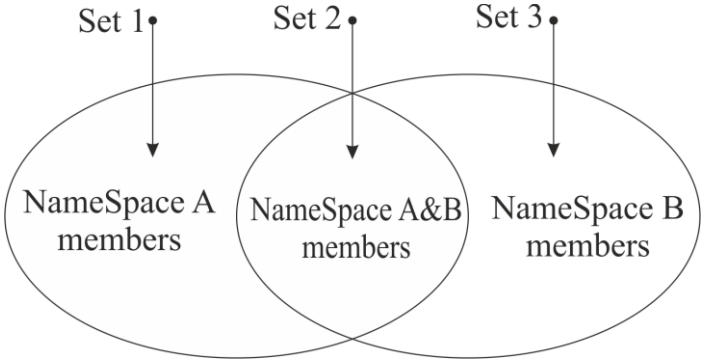


Рис. 4.1. Результат послідовного відкривання просторів *A* і *B*

На цьому рисунку позначено:

- множина 1 (*Set 1*) — її утворюють ідентифікатори, які належать простору A і не належать простору B;
- множина 2 (*Set 2*) — її утворюють ідентифікатори, які належать просторам *A* і *B*, тобто ідентифікатори, які перекиваються у іншому просторі;
- множина 3 (*Set 3*) — її утворюють ідентифікатори, які належать простору *B* і не належать простору *A*.

Після послідовного відкривання просторів *A* і *B* без використання оператора ОДД є доступними ідентифікатори За допомогою using-оголошення простір можна відкрити частково, тобто зробити доступним без оператора ОДД лише окремі його оголошення. Так, наприклад використання оператора using *A*:: і дає можливість надалі звертатись до змінної з простору імен *A* без оператора ОДД. Оператор using можуть використовуватись як в глобальній так і в локальній області дії. Але остання буде обмежувати дію цього оператора, наприклад

```
void F1()
{
using namespace A;
i = 1; // коректне звертання до A::i
}
void F2()
```

```
{
i = 1; // не коректне звертання до A::i
        }
void main()
{
F1();
i++; // не коректне звертання до A::i
A::i++; // коректне звертання до A::i
        }
```

Оператор using називають оператором перенесення імен із ПІ в поточну ОД. Його (а також і namespace) можна використовувати всередині іншого ПІ для того, щоб зробити в ньому доступними оголошення з іншої іменованої області, наприклад

Приклад № 4.6.

```
#include <iostream>
using namespace std;
namespace A
{
int i = 7;
}
namespace B
{
int i = 7, j = 7;
int f()
{
j = 0;
using A::i;
i = 1;
return B::j;
}
}
namespace C
{
using namespace A;
int j;
int f()
{
j = i = 2;
return C::j;
}
```

```
}
void main()
{
cout << " A::i = " << A::i << ", B::i = " << B::i <<
", B::j = " << B::f() << endl;
cout << " A::i = " << A::i << ", C::j = " <<
C::f() << endl;
        }
```

Як можна побачити відкривання простору A всередині простору C дало можливість доступу до змінної A::i без оператора ОДД. Спроба здійснити такі дії всередині простору B була також вдалою, незважаючи на те, що:

- по-перше простір A був лише частково (using A::i;) і локально відкритий (всередині функції B::f());
- по-друге мав однойменну змінну B::i. Незважаючи на те, що вона перекривала всередині простору B змінну A::i, усередині локальної області (в тілі B::f()) після часткового відкривання простору A утворилась ділянка з першочерговим доступом до змінної A::i.

Простір імен може бути вкладеним, наприклад

Приклад № 4.7.

```
#include <iostream>
namespace A // оголошення зовнішнього простору
{
int z = 2; // оголошення функції у A
void f(); // оголошення функції у A
namespace B // оголошення вкладеного простору
{
int k; // оголошення змінної у B
void f(); // оголошення функції у B
}
}
void A::f() // реалізація функції із A
{
z = 1; // звертання до змінної із A
}
void A::B::f() // реалізація функції із A::B
{
z++; // звертання до змінної із A
k = 2; // звертання до змінної із B
}
void main()
{
```

```
A::f(); // виклик функції із A
A::B::f(); // виклик функції із A::B
using namespace A; // відкривання простору A
z--; B::k--; // звертання до членів після відкривання
using namespace B; // відкривання вкладеного простору A::B
// звертання після відкривання A::B до членів, які належать
// обом просторам і не потрапляють в множину їх перетину
std::cout << " A::z = " << z << ", B::k = " << k << std::endl;
}
```

Наведені коментарі розкривають усі основні особливості програми. Але варто відзначити таке. У наведеній програмі вкладений простір *B* знаходиться в області дії, яка визначається зовнішнім простором *A*. А тому усі ідентифікатори простору *A* є доступними у тілі вкладеного простору *B* без використання додаткових засобів доступу. Тому потреби відкривати зовнішній простір у внутрішньому немає, але можна, наприклад

```
namespace A
{
int i = 20;
namespace B
{
using namespace A;
int f()
{
return i;
}
int i = 100;
int j = 30;
}
        }
```

При цьому в тілі функції *A*::*B*::*f*() буде використовуватись змінна *A*::*i*. У випадку, коли ім'я ПІ є достатньо великим за довжиною, допускається введення псевдонімів. Псевдоімена для просторів імен мають окрему назву — *аліаси* (від англ. *namespaxe alises*). Наведемо наприклад

Приклад № 4.8.

```
#include <iostream.h>
namespace Aaa
{
int i;
```

```
}
namespace A = Aaa; // оголошення псевдоніма
void main()
{
A::i = 7; // використання псевдоніма
std::cin >> Aaa::i;
        }
```

Загалом стандарт *C++11* визначає три способи визначення аліасів

```
namespace alias_name = ns_name;
namespace alias_name = ::ns_name;
namespace alias_name = nested_name::ns_name;
```

Область видимості аліаса визначається правилами, які висуваються до ОВ. Поряд з іменованими існує окремий вид ПІ анонімний простір імен (no-named namespaces, АПІ) або неіменований ПІ. Такий ПІ дає змогу створювати ідентифікатори (та декларації), які є унікальними всередині деякого файла і невідомі поза межами ОД файла. Таким чином, всередині файла, який містить анонімний ПІ, до його членів можна звертатись без використанні оператора дозволу дії (operator::), або за допомогою порожнього оператора дозволу дії. Поза межами файла ці ідентифікатори є недоступними, наприклад

Приклад № 4.9.

Файл - var.cpp

```
#include "var.h"
namespace // визначення АПІ
{
int i = 6;
int& f()
{
return i; // звертання до змінної з АПІ
}
}
namespace B // визначення ПІ - В
{
int f()
{
i++; // звертання до змінної з АПІ без явного
// використання ОДД
```

```
return ::f(); // звертання до функції з АПІ
// в межах файлу
// використовується порожній ОДД
// функція призначена
// для організації доступу
// до змінної з АПІ у інших файлах
}
}
```

Файл - var.h

```
#ifndef VARH
#define VARH
/* namespace // явна декларація АПІ -
// не відкриває доступу до АПІ
{
        extern int& i;
int f();
} */
namespace B // явна декларація ПІ - В
{
int f();
}
#endif
```

Файл - ex11_VS_2010.cpp

```
#include <iostream>
#include <typeinfo>
#include "var.h"
using namespace std;
namespace // визначення нового АПІ
{
int j = 5;
}
namespace A // визначення нового ПІ - В
{
int j = 6;
int f()
{
j = ::j + 1; // звертання до змінної з АПІ та ПІ
return j; // звертання до змінної з ПІ
```

```
}
}
void main ()
{
// ::i = 100; ::f(); - доступ заборонений
std::cout << " A::j = " << A::f() << endl << " Anonymous::i =
" << B::f() << std::endl << " Anonymous::j = " << ::j ;
}
```

У наведеному коді визначено у анонімний простори визначено у двох файлах var.cpp і run.cpp. Ідентифікатори АПІ на відміну від ідентифікаторів іменованих П (наприклад В), доступні лише в межах файлів, у яких вони визначені. Тому доступу до доступу до ідентифікаторів ::*i* та ::*f*() з АІП, який визначений у файлі var.cpp, немає. Навіть явно декларація простору у ФЗ var.h не дає можливість обійти це обмеження. Тому для його подолання використовується функція *B*::*f*() з іменованого ПІ — *B*. Користувацький ПІ завжди вважається вкладений у глобальну область видимості. Це вкладення допускає перекривання ідентифікаторів з ГОВПІ. У випадку такого перекривання, ще й якщо ідентифікатори виражають такі сутності як змінні, виникає колізія з поняттям ОД. Це пояснюється тим, що, незважаючи на те, що простори імен є глобальними конструкціями, область дії ідентифікатора з іменованого ПІ не збігається з ОД змінної з таким же імен, яка оголошена у глобальному просторі. У такому випадку зовнішня змінна (подібно до ідентифікатора будь-якої іншої сутності) у вкладеному блоці є невидимою, хоча останній і входить до області її дії.

Використовуючи порожній ОДД можна забезпечити доступ до змінної (або ідентифікатора) з глобальної області, наприклад

Приклад № 4.10.

```
#include <iostream> using namespace std;
int i = 5;       // Область дії файл
struct f{};      // Область видимості - ГОВПІ
namespace A
{
int i = 5;       // Область дії - файл
int f() // Перекривання ідентифікатора з ГОВПІ
{
i = ::i + 1; // звертання до змінної з ГО та ПІ
return i;       // звертання до змінної з ПІ
}
}
void main()
```

```
{
cout   << " A::i = " << A::f() << ", Global::i = " << i << endl;
using namespace A;
cout   << " A::i = " << A::i <<
", Global::i = " << i << endl;
::f F; // використання типу з глобального простору
}
```

Серйозна проблема виникає, коли одночасно використовується глобальний, анонімний та іменований простори імен, наприклад

Приклад № 4.11.

```
#include <iostream.h>
using namespace std;
namespace          // Ідентифікатор АПІ
{
int i = 4;
}
int i = 5;              // Ідентифікатор ГОВПІ
namespace A
{
int i = 6;              // Ідентифікатор ІПІ
}
void main()
{
A::i++;         // доступ дозволений
// i++;// доступ недозволений
::i++;  // доступ дозволений
cout   << " A::i = " << A::i << ", Global::i = " << i << endl;
}
```

У наведеному коді ідентифікатор АПІ, який перекривається ідентифікатором ГОВПІ, перестає бути доступним. Це пояснюється тим, що перевага використання порожнього ОДД віддається на користь ідентифікатора ГОВПІ. А тому у наведеному прикладі доступ до змінної і з АПІ є забороненим.

Анонімний простір може бути вкладеним. У цьому випадку ідентифікатори АПІ є відкритими у зовнішньому просторі. Доступ до них є можливим через використання ОДД з іменем зовнішнього простору, наприклад

```
namespace My
{
namespace
```

```
{
int f()
{
return 2;
}
int F()
{
return f();
}
}
}
...
int i = My::f();
```

Простори імен можна розширювати. Це означає, що оголошення простору не обов'язково повинно бути неперервним. Більше того, розширення ПІ може бути в іншому файлі. Цю властивість називають поширенням простору або багато- кратним оголошенням ПІ, наприклад

Приклад № 4.12.

Файл var.cpp

```
#include "var.h"

namespace A                    void out();
{                 {
int i = 7;                std::cout << "Namespace A" << std::endl;
void f()                  }
{         }

i = 6;
}
}
```

Файл - var.h

```
#ifndef VARH
#define VARH
namespace A
{
extern int i; void f();
}
#endif
```

```
Файл - ex_14_VS_2010.cpp
#include <iostream>
#include <typeinfo>
#include "var.h"
namespace A
{
int z = 7;
}
void main ()
{
A::f();
std::cout << " A::i = " << A::i << ", A::z = " << A::z
<< std::endl;
}
```

Технологія поширення простору імен дає можливість іншим способом реалізувати рознесення визначення ПІ на інтерфейсну частину та реалізацію. При використанні технології розширення ПІ файли var.h а var.cpp могли б мати такий вид

```
Файл - var.cpp
#include "var.h"
#include <iostream> namespace A
{

Файл - var.h
#ifndef VARH
#define VARH
namespace A
{
void out();
}
#endif
```

У C++11 можна оголошувати вбудовувані простори імен inline namespaces. Вбудований спефікатор inline робить декларацію вкладеного простору імен у точності так, ніби його ідентифікатор були оголошені у зовнішньому просторі імен. У першу чергу такі простори, призначені для підтримки еволюційного коду, який базується на версійному підході. Наприклад

```
#include <iostream> using namespace std; namespace My
{
```

```
#ifdef  cplusplus
inline namespace version_new
{
int f() { return 2;}
}
#else
inline namespace version_old
{
int f() { return 1;}
}
#endif
inline namespace nested_namespace
{
int f(int i) { return i;}
}
}
...
cout << My::f() << endl << My::f(3) << endl <<
My::nested_namespace::f(4) << endl;
```

Першим важливим висновком із наведеного прикладу є те, що доступати до ідентифікаторів з вбудованого простору імен можна безпосередньо - My::f(3), або використовуючи ідентифікатор вкладеного простору My::nested_ namespace::f(4).

Другим важливим висновком є можливість створення механізму управління версіями функції int f() в залежності від компілятора.

Об'єкти стандартної бібліотеки C++ (C++ standart library) визначені у власному просторі імен, який називається std. Це означає, що для того, щоб використовувати потокову систему вводу/виводу C++ після директиви її підмикання

```
#include <iostream.h>
треба було відкривати простір std
using namespace std;
```

або використовувати для звертання до об'єктів cout і cin іменований ОДД, наприклад std::cout і std::cin відповідно. Варто зазначити У залежності від реалізації бібліотечного файлу в одних випадках є необхідною явна декларація використання простору std, в інших випадках — така декларація не є обов'язковою. Зумовлено це тим, що в останньому випадку в бібліотечному файлі потокової системи вводу/виводу вже існує явна декларація відкривання простору імен std.

Лекція 5. Змінна. Типи об’єктів. Система простих типів. Директива typedef і визначення аліасів.

5.1. Поняття змінної

Змінні є програмними іменами, пов'язаними з областями пам'яті, яка виділяється для зберігання їх значень. Вони виступають фізичними суб'єктами у відкомпільованому коді, оскільки характеризуються виділеними на етапі виконання комірками пам'яті. Тут треба відзначити, що фізичний суб'єкт визначається валідною операцією взяття адреси &. А це означає, що пам'ять фізично повинна бути виділена завжди. Наприклад, у коді

```
#include <iostream>
using namespace std; double m[0];
struct A{} a; void main(void)
{
cout << "Mas: " << sizeof(m) << " " << sizeof(m[0]) << " "<< &m[0] <<" " << m << endl;
cout << "Ptr: "<< (double*)m <<" " << sizeof((double*)m)<< " "<< &m << endl;
cout << "Cl: " << sizeof(a) << " " << " " << &a;
}
```

незважаючи на нульовий розмір масиву та структури, пам'ять мінімального розміру (для вказівника він визначається типом, а для структури та класу — рівний одному байту) буде виділена. Цим забезпечується валідність операцій &m і &a.

Змінні, як фізичні суб'єкти, характеризуються:

- типом,
- областю дії (ОД),
- класом пам'яті.

Вони можуть вводитись двояко: згідно з стандартом C шляхом оголошення їх на початку програмного блоку, або згідно з стандартом C++ — у будь-якому місці до початку їх першого використання. У будь-якому випадку треба пам'ятати, що оголошення змінної породжує операцію, зокрема виділення пам'яті. І саме в цьому контексті, найпершим повинно розглядатись це оголошення.

На логічному рівні трактування коду оголошення змінної повинно трактуватись як послідовність двох дій: виділення пам'яті і накладання деякого логістичного інтерфейсу доступу до цієї пам'яті.

Цей інтерфейс, окрім власне доступу до пам'яті, забезпечує трактовку в контексті типу вмісту виділених комірок. Наприклад код

```
int i;
```

на логічному рівні повинен трактуватись як виділення (наприклад 4 байт) пам'яті, і вміст цієї пом'яті, якщо він вичитується за допомогою змінної і є цілим числом. Це означає, що:

- по-перше вміст виділеної комірки може бути прочитаний різними інтерфейсами,
- по-друге різні інтерфейси забезпечують різну трактовку вмісту вибраної ділянки пам'яті. Типовим прикладом описаного підходу є використання об'єднань.

У найпершій класифікації змінні поділяють на динамічні (вказівникові), статичні (не путати з класом пам'яті static) та локальні (автоматичні змінні локальної ОД).

Основною відмінністю між вказаними видами змінних є організація способу адресації. У випадку динамічних змінних використовується непряма адресація (за допомогою вказівника); у випадку статичних — пряма (за допомогою ідентифікатора змінної). Для статичних змінних попередньо (на етапі компіляції) відводиться пам'ять (статична пам'ять), оскільки наперед відомий їх розмір. На відміну від статичних для динамічних змінних пам'ять виділяється в процесі виконання програми (така пам'ять називається динамічною).

З точки зору розробника це означає у випадку динамічної змінної звертання до комірки пам'яті (яку у більшості випадків попередньо ще й треба виділити) через адресу, яка зберігається у іншій змінній (вказівнику чи посиланні). У випадку статичної змінної звертання до наперед виділеної комірки пам'яті відбувається за прямою адресою, яка уособлена ідентифікатором.

Ідентифікатор змінної є l-виразом. У якості оператора він повертає lvalue, яким виступає посилання на комірку пам'яті, що виділялась під змінну. Це означає, що правильним і, що більш важливо, функціональним є такий код

```
int x;
x;
```

Діями наведеного коду буде виділення блоку пам'яті розміром sizeof(int) та повернення посилання на нього.

З точки зору виконання програми непряма адресація сповільнює виконання програми, проте дозволяє формувати програмні об'єкти динамічно тобто в процесі виконання програми і, теоретично, довільного розміру.

Детальніше динамічні змінні розглядаються в наступному розділі. Розглянемо основні характеристики змінних.

5.2. Типи об'єктів

Під типом об'єкта (object type), зважаючи на стандарт C++-98, розуміється будь-який тип, який не є типом функції, посиланням або типом void. Класична категоризація типів передбачає існування двох категорій: прості типи та складені (compound types). За стандартом C++-98 прості типи називаються фундаментальними (fundamental types).

Цим же стандартом визначена ще одна категоризація типів: скалярні типи і типи класів. Скалярні типи включають прості типи, типи переліки і вказівникові типи. Типами класів є будь-які оголошення, які реалізуються за допомогою виконані за допомогою трьох ключових слів класів: class, struct або union, які визначають відповідно структуру, клас та об'єднання.

Окрім цього стандарт C++-98 ввів до розгляду поняття агрегату (aggregate). Ним вважається масив або клас, який не має користувацьких (визначених користувачем) конструкторів, закритих або захищених нестатичних (non static) членів-даних, базових класів і віртуальних функцій. Оскільки у цьому випадку отримується своєрідний масив (накопичення) елементів, то це породило термін «агрегат».

Агрегати можуть ініціалізуватись за допомогою ініціалізатора, наприклад:

```
struct A
{
short j; char s[3];
} a = { 10, {'1', '2','\0'} };
```

Типово агрегати мають тип POD, але це не обов'язково. Член-даних j агрегату A міг би розглядатись як змінна типу клас, якщо б він мав неявний конструктор із параметром цілого типу та конструктор копії.

Тип POD, який також визначається стандартом C++-98, як тип простих даних і є множиною, яка складається із скалярних типів, типів POD-struct, POD-union та масивів цих типів і їх версій із модифікаторами const і/або volatile. Абревіатура POD розшифровується як plain-old-data. Цей тип даних відіграє дуже важливу роль в мові C++. Його основним призначенням є забезпечення сумісності з відповідними типами мови C, тобто для зв'язку мов C і C++.

POD-тип трактується як тип із стандартним розміщення в пам'яті. Тому для його екземплярів допускається використання операцій прямого доступу до пам'яті, зокрема memcpy() та memset(). Додавання чи віднімання конструкторів не впливає на розміщення об'єктів у пам'яті.

POD-struct є клас-агрегат (агрегатний клас), який не має звичайних (нестатичних) не POD-членів і не має користувацьких операторів копіювання, присвоювання та конструктора/деструктора. Це означає, що звичайні члени даних у POD- struct не можуть адресувати інші члени, не POD-struct, не POD-union, масиви вказаних типів та посилання. Слід зазначити, що для визначення

класу-агрегат можна використовувати або ключове слово struct, або ключове слово class.

POD-union має таке ж визначення за винятком того, що замість зарезервованого слова struct використовується union.

У термінології тривіальних визначень клас /структура є типу простих даних, якщо вони або тривіальні, або із стандартним розміщенням і якщо типи усіх нестатичних членів також є типами простих даних.

Тривіальним називається клас, який містить лише:

- тривіальний конструктор по замовчуванні;
- тривіальний конструктор копіювання по замовчуванні;
- тривіальний оператор присвоєння копії;
- тривіальний деструктор, який не повинен бути віртуальним. Клас із стандартним розміщенням це клас, який:
- не містить нестатичних членів, які є типу класу (або масиву класів) із нестандартним розміщенням чи посилання. Тобто можуть мати тільки статичні члени не стандартних типів;
- не містить віртуальних функцій;
- не містить віртуальних базових класів;
- не мають батьківських класів із нестандартним розміщенням;
- не мають батьківських класів того ж типу, якого є перший нестатичний член-даних (якщо він є);
- мають один і той же специфікатор доступу (public, private, protected) для всіх нестатичних методів.

Варто відзначити ще декілька важливих зауважень, які мають відношення до типу POD:

- для POD-struct і, POD-union визначено макрос offsetof(). Для усіх решти типів результат формально вважається невизначеним;
- тип POD може входити в об'єднання. Це використовується для визначення обмежень для типів POD;
- статичний об'єкт типу POD, який ініціалізується константними виразами, буде ініціалізованим до входження у його програмний блок і до моменту динамічної ініціалізації будь-якого іншого об'єкта (необов'язково типу POD);
- вказівники на члени не є типами POD на відміну від вказівників на інші типи;
- типи POD-struct або POD-union можуть мати статичні члени, члени-typedef, вкладені типи і методи.

Для простоти розуміння POD-типами треба сприймати усі типи, які наслідувались з мови C і нові скалярні типи C++. Ц означає, що будь-який тип, який використовується з елементами ООП вже не є POD-класами. Відповідних стосовно змінних таких типів не можна здійснювати суджень про внутрішню структуру і фізичне розміщення у пам'яті. Наприклад структура:

```
 struct A
{
char b;
};
```

є POD-типом, а об'єкт структури

```
struct A
{
public:
char b;
        };
```

за стандартом мови не є POD-об'єктом, оскільки використано зарезервоване слово public. І стосовно нього формально забороняється використовувати дії на рівні внутрішнього представлення, на приклад застосувати функцію memset. Вказана заборона носить рекомендаційний характер. ЇЇ нехтування їде врозріз із вимогами стандарту.

Для перевірки належності типів до фундаменального чи тривіального типу, класу із стандартним розміщенням чи типу POD у бібілотеці C++ введено шаблони: template < class T > struct is_fundamental; template < class T > struct is_trivial, template < class T > struct is_standard_layout, template < class T > struct is_pod (ФЗ — <type_traits>) з перевантаженим оператором приведення типу до bool. Наведемо приклад використання цих шаблонів

Приклад № 5.1.

```
#include <iostream>
#include <type_traits>
using namespace std;
struct A
{
int m;
};
struct B
{
B() {}
};
struct C
{
int m1;
private:
int m2;
};
struct D
```

```
{
virtual void f() {}
};
void main(void)
{
cout << boolalpha;
cout << is_trivial< A >::value << '\n';
cout << is_trivial< B >::value << '\n';
cout << is_standard_layout< A >::value << '\n';
cout << is_standard_layout< C >::value << '\n';
cout << is_pod< A >::value << '\n';
cout << is_pod< D >::value << '\n';
cout << is_fundamental< A >::value << '\n';
cout << is_fundamental< int >::value << '\n';
cout << is_fundamental< int& >::value << '\n';
cout << is_fundamental< int* >::value << '\n';
        }
```

Бібліотека <type_traits> реалізована у C++11 і призначена для роботи із типами на етапі компіляції. Вона містить набір функціональностей у виді шаблонів, які забезпечують роботу з типами на етапі компіляції.

5.3. Система простих типів

Система простих типів (СПП) повністю підтримує усі базові типи ANSI C і розширює їх власними. ЇЇ логічна схема наведена на Рис. 5.1. До переліку наведених на Рис. 5.1 необхідно ще додати тип void.

СПП не допускає зміни типу змінної після її оголошення і складається з двох підкатегорій. Перші категорія - це інтегральні типи: bool, char, char16_t, char32_t, wchar_t, short, int, long, long long. Частина з них (див. Рис. 5.1) може бути модифікована за допомогою специфікаторів знаковості signed та безнаковості unsigned. Ці специфікатори можуть використовуються для представлення цілих значень (у ряді випадків модифікатори типу можуть розглядатись як імена основних типів, зокрема int.

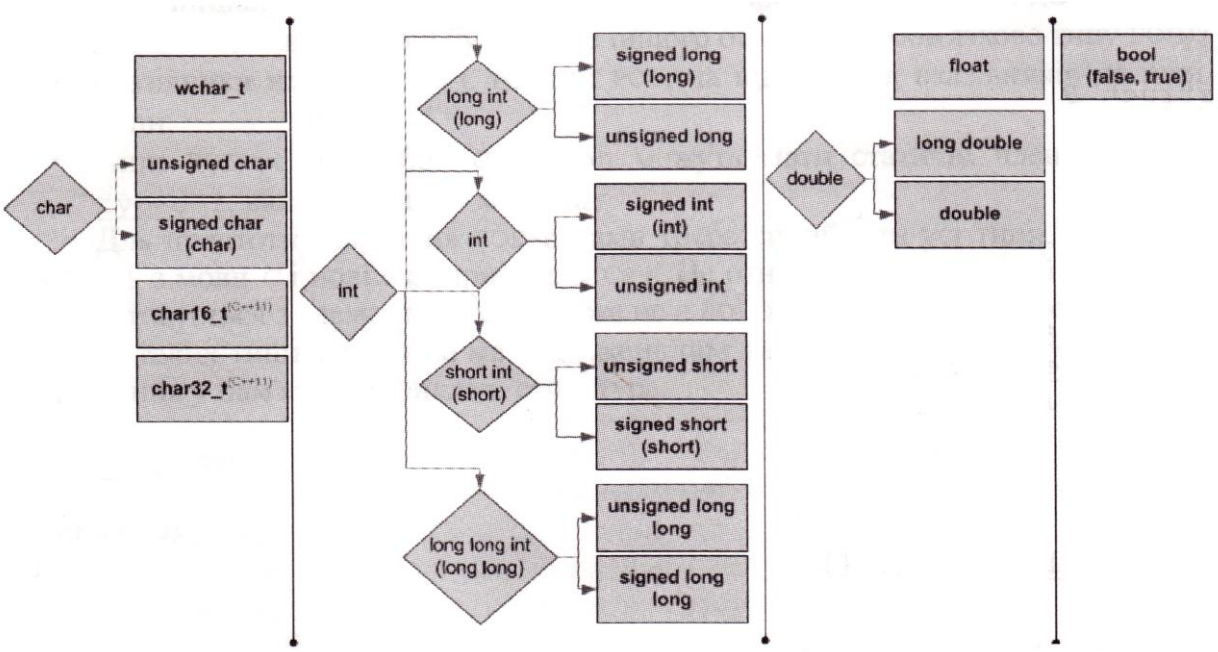


Рис. 5.1. Система простих типів C++

Загалом, якщо тип опущено, то вважається, що типом є int.

Друга категорія це типи з плаваючою крапкою. До них відносять такі типи: float, double, long double. У відповідність із стандартом IEEE-754 вони мають розрядність 32, 64 та 80 біт.

Бібліотека <type_traits>, подібно до випадку POD-типу, дає засоби перевірки на цю категоризацію (шаблони template<class T> struct is_integral, template< class T > struct is_floating_point і template<class T > struct is_arithmetic), наприклад

Приклад № 5.2.

```
#include <iostream>
#include <type_traits>
using namespace std;
class A {};
void main(void)
{
cout << boolalpha
<< " is_integral< A > - " << is_integral< A >::value << '\n' <<
"        is_integral< float > - " << is_integral< float >::value << '\n' <<
"        is_integral< int > - " << is_integral< int >::value << '\n'
<< " is_floating_point< A > - " << is_floating_point< A
>::value << '\n' <<
"        is_floating_point< float > - " << is_floating_point< float
>::value << '\n' <<
"        is_floating_point< int > - " << is_floating_point<   int
```

```
>::value << '\n'
<< " is_arithmetic< A >     - " << is_arithmetic< A >::value
<< '\n' <<
"         is_arithmetic< int > - " << is_arithmetic< int  >::value
<< '\n' <<
"         is_arithmetic< int& > - " << is_arithmetic< int& >::value
<< '\n' <<
"         is_arithmetic< int* > - " << is_arithmetic< int* >::value
<< '\n' <<
"         is_arithmetic< float > - " << is_arithmetic< float >::value
<< '\n' <<
"         is_arithmetic< float&> - " << is_arithmetic< float&>::value
<< '\n' <<
"         is_arithmetic< float*> - " << is_arithmetic< float*>::value
<< '\n';
}
```

У мові C (та й C++) розміри простих ти визначався не стандартом, а релізом компілятора і обраним операційним середовищем. Гарантувалось стандартом лише виконання правила

```
1 == sizeof(char) <= sizeof(short) <=sizeof(int) <=
        sizeof(long) <= sizeof(long long).
```

У залежності від розмірності типу змінними були діапазони значень, які могли адресуватись змінними цього типу. Відповідно розробка програми повинна була бути максимально незалежною від розміру типу і фіксованих значень. Для отримання інформації про розмір типу використовувались функція sizeof(), яка повертала значення розміру у байтах, та бібліотека макросів <cstdint>, яка в основному визначалась у ФЗ <climits>, <cfloat>, <cstdint>, яка у вигляді констант повертала усю інформацію про прості типи, зокрема мінімальні та максимально допустимі значення.

Для вирішення описаної проблеми у C++11 зроблено таке:

- вироблені рекомендації мінімальної розрядності типів в залежності від обчислювального середовища;
- визначені діапазони допустимих значень в залежності від використовуваної арифметики та розрядності;
- у ФЗ <cstdint> розширено набір цілих типів та макросів;
- введено бібліотеку <limits>, а у ній — шаблон numeric_limits, який дає можливість отримати усю інформацію про тип.

Наведемо приклад використання шаблону numeric_limits

Приклад № 5.3.

```
#include <limits>
#include <iostream>
using namespace std;
void main(void)
{
cout << "type\tlowest\thighest\n"; cout << "int\t"
<< numeric_limits< int >::lowest() << '\t'
<< numeric_limits< int >::max() << '\n'; cout << "float\t"
<< numeric_limits< float >::lowest() << '\t'
<< numeric_limits< float >::max() << '\n'; cout << "double\t"
<< numeric_limits< double >::lowest() << '\t'
<< numeric_limits< double >::max() << '\n';


int i; double x = 0;
i = (int)x; i = int(x);
```

Більш складні операції пов'язані із опціями над типами вирішуються шляхом використання підсистема RTTI, якій надалі присвячено окремий параграф.

Для змінних різних типів у C++ введено новий формат приведення типів: type(variable) = (type)variable. Це означає, що два наступні приведення типів є ідентичними:

```
int i; double x = 0;
i = (int)x; i = int(x);
```

У мові C++, подібно до C чи більшості інших мовах програмування, сповідувався стандартний підхід до типізації змінної, а саме — тип змінної повинен бути вказаний у явному вигляді. Проте, після появи технологій мета програмування, в деяких випадках тип у явній формі не завжди може бути заданий. Це ускладнює процес зберігання проміжних значень у комірках пам'яті, які виділялись під змінні програми. Для вирішення цієї проблеми в стандарті C++11 пропонується два способи.

За першим з них пропонується використовувати зарезервоване слово auto до змінної, яка ініціалізується у явній формі, наприклад

```
auto x = 0;
```

Це призведе до того, що тип змінної буде визначений компілятором в процесі семантичного аналізу за ініціалізуючим значенням (значенням

ініціалізатором) рівним 1. У цьому випадку на практиці дуже часто використовується термін неявне визначення типу змінної x. За положеннями теорії проектування навіть пропонується вважати, що x є сутністю, яка виражає змінну.

Очевидно, що у наведеному прикладі використання зарезервованого слова auto для визначення змінної x можна уникнути, оскільки тип змінної x є очевидним із значення літеральної константи. Проте, ініціалізуюче значення може бути не настільки очевидним, наприклад

```
#include <vector>
#include <algorithm>
#include <iostream>

bool greater5 ( int value )
{
return value > 5;
}
...
std::vector <int> v;
auto x =
std::stable_partition( v.begin(), v.end(), greater5 );
```

У наведеному прикладі значення змінної x визначається значенням, яке поверне екземпляр шаблонної функція stable_partition. Сам екземпляр, у свою чергу визначатиметься спеціалізацією узагальненого типу шаблону stable_partition.

Використання auto іноді може суттєво зменшити надлишковість коду. Наприклад, код

```
for (vector <int>::const_iterator it = v.begin();
it != v.end(); ++it);
```

можна записати у вигляді

```
for (auto it = v.begin(); it != v.end(); ++it);
```

Використання зарезервованого слова auto надає можливість в одному оголошенні створювати змінні різних типів, наприклад

```
auto i = 12,
j = -123.456,
a = "hello auto literal",
```

```
b = std::string( "hello auto string" ),
        my = some_my_type();

"int*" : " n't correctly defined" ) << endl;
}
```

Допускається також композиційне використання зарезервованого слова auto, наприклад

```
double k;
const auto *p = &k;         // тип p є const double*
auto const *p2 = &k;        // тип p є double const*
```

Поєднання використання auto і оператора new забезпечить інціалізацію та динамічне виділення пам'яті, наприклад

Приклад № 5.4.

```
#include <iostream>
#include <typeinfo> using namespace std;
void main(void)
{
struct A { } a;
auto ap = new auto ( a );
cout << "type is " << ( (typeid( A* ) == typeid( ap ) )? "A*"
: " n't correctly defined" ) <<endl; auto dp = new auto( -23.56 );
cout << "type is " << ( (typeid( double* ) == typeid( dp )
) ? "double*" : " n't correctly defined" ) << endl; auto fp = new auto( 3.5f );
cout << "type is " << ( (typeid( float* ) == typeid( fp ) ) ?
"float*" : " n't correctly defined" ) << endl; auto* ip = new auto( 235 );
cout << "type is " <<        ( (typeid( int* ) == typeid( ip ) ) ?
"int*" : " n't correctly defined" ) << endl;
}
```

Тут варто звернути увагу на коректність визначення типів змінних *ap*, *dp*, *fp* та *ip*. В усіх випадках використовувася оператор new, і в усіх випадках компілятор типом вказаних змінних вибрав не void*, а саме як вказівник на відповідний тип. Наприклад для випадку змінної *ap*, типом обрано вказівник на структуру *A*: *A**. За другим способом тип змінної може бути визначений при допомозі зарезервованого слова decltype із типу вже існуючої змінної, наприклад

Приклад № 5.5.

```
#include <iostream>
```

```
#include <typeinfo>
int f()
{
return 1;
}
void main(void)
{
auto x = 1.1, y = 1;
decltype(x) p = 1; // тип - double
decltype(1.1) p1; // тип - double
// оскільки 1.1 є rvalue)
decltype(new int[10]) p2; // тип - int *
decltype(f()) p3; // тип - int
decltype((x)) p4 = y; // тип - double&
// оскільки (x) є lvalue)
decltype(&x) p5; // тип - double *
std::cout << typeid(p).name() << typeid(p1).name() <<
typeid(p2).name() << typeid(p3).name() <<
typeid(p4).name() << typeid(p5).name();
        }
```

Змінна, яка визначає тип новостворюваної змінної, повинна існувати до виклику decltype. Це означає, що її можна задати у довільний спосіб - як звичайну змінна, літеральну константу, тощо.

У наведено прикладі тип змінної x буде double, оскільки визначатиметься за допомогою зарезервованого слова auto із літерального значення 1.1. Для змінної p тип буде визначатись не за значенням ініціалізатора (у прикладі це ціле значення 1), а типом змінної x. Тобто тип змінної p буде також рівним double. Значення ініціалізатора не є обов'язковим при використанні decltype окрім випадків, коли початкове присвоєння значень вимагається правилами мови. Це означає, що у попередньому коді можна було записати decltype(x) p;. І це б не вплинуло на визначення типу змінної p. Якщо б змінна x була визначена константою (const double x = 1.1;), то ініціалізація змінної p вже є обов'язковою, оскільки правила мови C++ вимагають негайної ініціалізацію константної змінної при її оголошенні. У Прикл. № 5.5 вимога негайної ініціалізації висувається до змінної p4, оскільки її типом є посилання.

Використання decltype є найбільш ефективним при використання із змінними, тип яких визначається через auto. Це зумовлено тим, що тип визначений за допомогою auto не є очевидним, але є відомим компілятору. Стосовно змінних, тип яких задається явно, використання decltype є менш ефективним. Тому у цьому випадку цей оператор не рекомендується використовувати оскільки знижується читабельність програмного коду.

Тип, який визначається за допомогою auto може відрізнятись від того, який визначається за допомогою decltype. Наприклад

Приклад № 5.6.

```
#include <iostream>
#include <typeinfo>
using namespace std;
void main(void)
{
int i = -1;
char * p = new (&i) char[4]; auto x = p[0];
decltype( (p[0]) ) y = p[1]; x = y = 8;
cout << "Type of p : " << typeid(p).name() << endl << "Type of x : " << typeid(x).name() <<
endl << "Type of y : " << typeid(y).name() << endl;
cout << " p[0] : " << int( p[0] ) << ", p[1] : " << int( p[1] );
}
```

У наведеному прикладі змінна *x* є типу char. У той же час тип змінної *y* є char& і ця змінна посилається на комірку *p*[0]. Відповідно записати туди значення стало можливим через два інтерфейси: *p*[0] і *y*.

5.4. Директива typedef і визначення аліасів

Директива typedef є інструкцією абстрактного рівня, яка використовується для визначання ідентифікаторів, які, у свою чергу, є псевдоіменами (псевдонімами, іншими назвами, іншими іменами) системних (вбудованих) або користувацьких типів (у тому числі вказівників на функції) та просторів імен. Ці псевдоімена можуть використовуватись у якості зарезервованих слів.

Абстрактний рівень директиви typedef визначає область її використання. Це означає, що директиву не можна використовувати на рівні абстракції шаблонів та на фізичному рівні змінних.

Синтаксис використання директиви typedef є таким

```
typedef имя_типу идентификатор_псевдонім;
```

або

```
typedef визначення_користувацького_типу идентификатор_псевдонім;
```

Наведемо приклад використання кожного із способів

```
typedef char* PCHAR;     // 1-й спосіб
```

```
typedef enum {f, t} boolean;     // 2-й спосіб
typedef enum B {fl, tr} BOOLEAN;       // 2-й спосіб
struct AA {}; // побудова типу
typedef AA A;        // 1-й спосіб
// Використання псведоімен у якості ідентифікаторів типів
PCHAR p1;   - еквівалентне оголошенню - char* p1;
boolean b;   - еквівалентного оголошення немає
B b1;  - еквівалентне оголошенню - BOOLEAN b1;
A a;    - еквівалентне оголошенню - AA a;
```

За другим способом код з директивою typedef перед введенням псевдоімені ще й визначає користувацький тип. У одному випадку він є анонімним, а у іншому — має ім’я B. це означає, що у першому випадку використання типу можливе лише шляхом використання псевдоімені, а у другому — використанням не лише псевдоімені, а й імені типу B. Треба розуміти, що сама директива користувацький тип не вводить. Її призначенням є виключно визначення псевдоімені. А засоби мови дозволяють об'єднати ці дві дії.

Директива typedef може використовуватись у локальній і глобальній областях дії, визначаючи при цьому власну область видимості для своїх ідентифікаторів. Всередині цієї області областей кожен ідентифікатор, який визначений директивою, у кожному місці використання є синтаксично еквівалентним зарезервованому слову, яке є іменем типу, або імені користувацького типу. Наприклад

```
typedef unsigned char UINT;     // глобальне псевдоім'я (ГПІ)
// вбудованого типу char
void F()
{
typedef unsigned int UI; // локальне псевдоім'я
typedef unsigned char UC; // локальне псевдоім'я
UI i = 0;
UC c = (char) 0;
UINT j = 0;   // правильне використання ГПІ
}

void main()
{
F();
UINT z = 0;                  // правильне використання ГПІ
char k = (char) 0;           // імена вбудованих типів
                             // не перекриваються і є доступними
//UC u = (char)0;            // локальне псевдоім'я не можна
```

```
//UI y = 0;                    // використовувати за межами ОВ
}
```

У наведеному коді коментарі повністю виражають пояснення використання директиви typedef у локальній та глобальній областях дії.

В одній декларації можна оголошувати декілька псевдонімів, наприклад

```
typedef char CH, *PCHAR;
...
PCHAR s = new CH[10];
```

Директива typedef не є виконавчою інструкції, а тому може використовувати у ФЗ беж жодних застережень. Цільовим призначенням директиви є мінімізація довжин імен і забезпечення типізації параметрів у завданнях, насамперед, заміщення локальних параметрів, наприклад

Приклад № 5.7.

```
#include <iostream>
#include <vector>
typedef std::vector<int>::iterator VIT;
VIT it;
typedef int (*FBIN)(int,int);
int start(FBIN f, int a, int b)
{
return f(a, b);
}
int max(int a, int b)
{
return ( a > b ) ? a : b;
}
int add(int a, int b)
{
return a + b;
        }
int mult(int a, int b)
{
return a * b;
}
void main(void)
{
std::cout << "Max (1,2) : " << start( max, 1, 2 ) << std::endl;
std::cout << "Add (1,2) : " << start( add, 1, 2 ) << std::endl;
```

```
std::cout << "Mult (1,2) : " << start( mult, 1, 2 ) << std::endl;
}
```

У наведеному коді:

• ідентифікатор *VIT* використовувався для мінімізації використання дуже довгого імені типу std::*vector*<int>::*iterator*;

• ідентифікатор *FBIN* використовувався для формування типу параметрів функції *start*.

Стандарт *C++11* дозволяє використовувати оператор using замість директиви typedef. У цьому випадку псевдоімена називаються аліасами. Загальний синтаксис визначення аліасів є таким

```
using identifier = type_name;
```

Наприклад замість оголошення

```
typedef int (*FBIN)(int,int);
```

можна записати

```
using FBIN = int (*)(int,int);
```

Цільовим призначення аліасів є спрощення роботи з шаблонами.

Лекція 6. Змінна. Область дії. Класи пам’яті. Поняття ініціалізатора. Життєвий цикл змінної.

6.1. Область дії

Областю дії (ОД) змінної як фізичної сутності може бути:

- програмний блок;
- тіло функції/шаблону функції;
- файл;
- клас/шаблон класу.

На відміну від області видимості, поняття якої стосується ідентифікатора, область дії стосується виключно фізичної сутності, тобто змінної.

На Рис. 6.1 показано відображення області видимості для випадку вираження ідентифікатором такої сутності як змінна. Незважаючи на існування наведеного на рисунку взаємного відображення, треба розуміти, що ці характеристики є різними і призначені для використання стосовно необов'язково однакових сутностей.

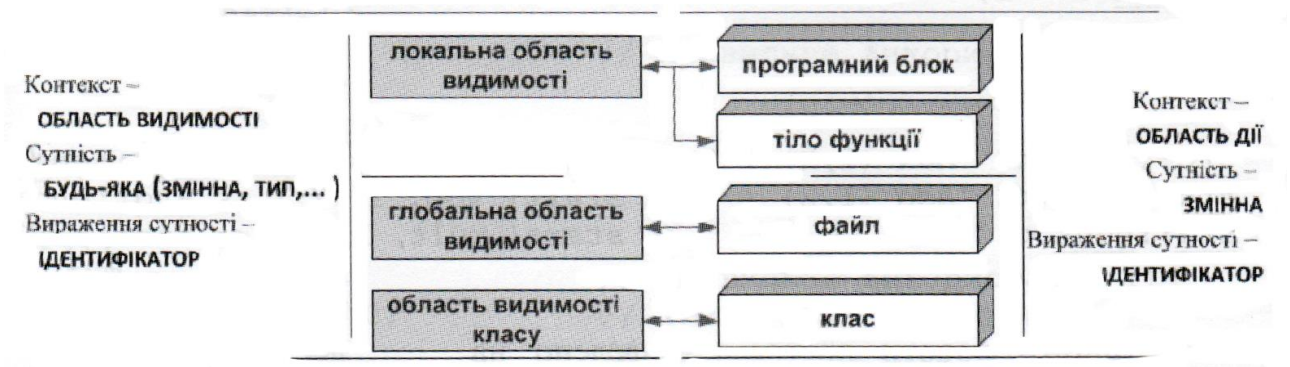


Рис. 6.1. Взаємне відображення області видимості та області дії

ОД змінної визначається концепцією програмного блоку (ПБ). Змінна, оголошена в програмному блоці називається локальною і є видимою усюди в області, яка визначається початком оголошення і закінчення програмного блоку. ЇЇ можна використовувати в усіх подальших вкладених блоках. При цьому оголошення змінної з тим самим іменем у вкладеному блоці приховає її значення, яке вона мала у зовнішньому блоці. Після повернення управління у зовнішній блок змінній повернеться її попереднє значення. Наприклад

Приклад № 6.1.

```
#include <iostream>
using namespace std;
void main(void)
{
```

```
int z = 5;
{
cout << z;
int z = 4;
cout << z;
}
cout << z;
        }
```

У мові *C* описана поведінка коду називалась *технологією приховуванням ідентифікаторів*. Це не зовсім відповідало дійсності. Так у Прикл. № 6.1 ідентифікатор *z* вкладеного блоку справді перекриває однойменний ідентифікатор зовнішнього блоку. Але треба пам'ятати, що ці ідентифікатори надають доступ до зовсім різних комірок пам'яті. Тому перекривання має місце виключно на рівні логістики етапу розробки програмного забезпечення ОД змінної може бути увесь файл. Така змінна оголошуються поза блоками і класами і називається глобальною. Вона може бути:

- глобальною — доступною у даному та інших файлів проекту,
- статичною глобальною — доступною лише у даному файлі,
- зовнішньою — доступною з інших файлів проекту.

Наведено приклад оголошення таких змінних

```
int number;                 // глобальна змінна
extern int number;          // зовнішня змінна
static int number;          // статична глобальна змінна,
                            // яка є закритою (тобто локальною) з
                            // точки зору
                            // доступу до неї з інших файлів проекту
```

ОД змінної може бути функція або прототип функції. Така змінна також називається локальною, а її ОД визначається початком оголошення у прототипі, заголовку чи тілі функції та кінцем прототипу або тіла функції. Наприклад:

```
int A (int x); // локальна змінна y прототипі
int B (int y)
{
x++; // локальна змінна x недоступною
return x; // локальна змінна y є доступна
}
int A (int x)
{
x++; // локальна змінна є доступна
z++; // локальна змінна є недоступною
```

```
int z;
return z; // локальна змінна z є доступна
}
```

Глобальна змінна, на відміну від локальної, по замовчуванні ініціалізується "нульовим" значення, якщо у типі, за яким вона визначена, таке значення існує. Але тут треба пам'ятати, що явна ініціалізація впливає на дію модифікатора extern, що може призвести до помилки періоду зв'язування.

Сучасні рекомендації по програмуванню радять мінімізовувати кількість глобальних змінних, які є причиною постійних витрат пам'яті. Саме з цих міркування такі сучасні мови, як наприклад Java, мінімально (на рівні статичних змінних) підтримують технологію глобальних змінних.

І, нарешті, ОД змінної може бути клас (структура C++) чи іменована область — простір імен.

6.2. Класи пам'яті

Клас пам'яті (КП) визначає метод, за яким компілятор зобов'язаний виділяти пам'ять для зберігання змінної. Існують чотири основні класи пам’яті: auto, extern, static, register, thread_local52 та два модифікатори const та volatile (у класах ще й mutable). Класифікація класів пам'яті наведена на Рис. 6.2.

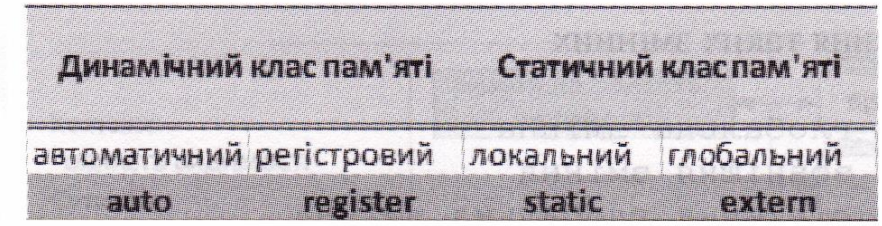


Рис. 6.2. Класифікація класів пам'яті

Клас пам'яті auto використовується для оголошення змінних, які з коміркою пам'яті зв'язуються динамічно (тобто під час виконання операторів оголошення), а з типом — статично. Такі зміні називаються *автоматичними*, оскільки вони створюються і знищуються без втручання розробника – програміста і розміщуються в *локальній пам'яті* (пам'яті, яка розподіляється автоматично). За виключенням адреси, усі атрибути автоматичних змінних зв'язуються статично. Виключення становлять об'єднання, які у різні моменти можуть містити зміні різних типів, а тому зв'язуються з типом динамічно. Використовуються автоматичні змінні в операторах, які оголошуються в тілі функцій чи всередині інших блоків операторів. Об'єкти, які оголошені з класом пам'яті імена auto, розміщуються в локальній пам'яті безпосередньо перед виконанням функції чи блока операторів. При виході з блока, або після повернення з тіла функції відповідна локальна пам'ять звільняється і усі раніш розміщені в ній об'єкти знищуються. Таким чином модифікатор впливає на тривалість життя змінної. У явній формі модифікатор auto використовується

рідко, оскільки усі змінні, які оголошуються безпосередньо в тілі функціях чи операторному блоці по замовчувані розміщуються в локальній пам'яті. Тому явного використання модифікатора auto для таких змінних є зайвим. Поза меж блоку чи функції цей модифікатор не використовується.

На противагу автоматичним *зовнішні* змінні, які визначаються класом пам'яті extern, зберігають свої значення постійно. По замовчуванню зовнішні змінні з однаковими іменами у різних файлах посилаються на один і той самий програмний об'єкт (технологія *external linkage*).

Основною відмінністю зовнішніх змінних є те, що оголошуються вони поза межами функцій, а тому їх ОД є проміжок, який визначається точкою оголошення і кінцем файла. Такі змінні при відсутності явної ініціалізації приймають нульові значення (тут необхідно розрізняти процеси оголошення, при якому не виділяється пам'ять, і ініціалізації, при якому виділяється пам'ять). Означену зовнішню змінну називатимемо *глобальною*.

Якщо треба зробити посилання на зовнішню змінну визначену в іншому файлі, то її оголошення повинно містити декларацію extern

```
extern int i;          //оголошення зовнішньої змінної з іншого файла
extern int m[];        //оголошення зовнішнього масиву з іншого файла
```

При цьому створення (виділення пам'яті) та ініціалізація змінної і повинна бути здійснена у сторонньому файлі. У цьому файлі можливими є два варіанти побудови змінної i чи m. За першим з них у явній формі використовується декларація extern та ініціалізація

```
// створення+ініціалізація зовнішньої змінної
extern int i = 1;
// створення+ініціалізація зовнішнього масиву
extern int m[10] = {0};
```

За другим варіантом достатньо звичайного оголошення глобальної змінної, яке може навіть не містити явного оголошення. У цьому випадку матимемо ініціалізацію нульовим значенням по замовчуванні, незалежно від того якого типу є змінна: простого чи складеного. Наприклад

```
int i = 1;
int m[10];    // кожна комірка масиву m буде ініціалізована нулем
```

Другий варіант означає, що відсутність модифікатора extern при оголошенні глобальної змінної автоматично її робить доступною назовні. Проте в обох варіантах оголошення та ініціалізація змінної повинні розглядатись як виконавча дія. За положеннями загальної теорії ООП такі оголошення є

створенням глобальної сутності, яке спричинене попереднім викликом і роботою алокаторів та ініціалізаторів комірок пам'яті в глобальній області.

Організація extern-декларацій необхідна також випадку, коли задіюються змінні, ініціалізація яких відбувається після використання, наприклад

Приклад № 6.2.

```
#include <iostream>
using namespace std;
void main (void)
{
extern int i; // локальна декларація
i++;
// extern int j = 2; - помилка -
// створення та ініціалізація зовнішньої змінної
// у локальній ОД
cout << "i = " << i;
}
int i = 5;
```

Клас пам'яті static визначає змінні, які зв'язуються з коміркою пам'яті на етапі компіляції і залишаються з нею зв'язаною до завершення програми. Використовується для створення недоступних за межами ОД змінних, під які виділена постійна пам'ять. Такі змінні будемо називати статичними. Якщо ОД такої змінної є файл, то вона називатиметься глобальною статичною або просто статичною; якщо ОД є функція, то — локальною статичною.

Глобальна статична змінна строюється за схемою глобальних об'єктів перед виконанням start-up функції. ОД такої змінної є виключно файл, у якому вона оголошена. Це означає, що вона є невидимою з інших файлів проекту.

Локальна статична змінна створюється при першому виклику функції і зберігає своє значення між викликами функції. Проте доступна така змінна лише в межах своєї ОД — функції чи програмного блоку, наприклад

Приклад № 6.3.

```
#include <iostream>
using namespace std;
void f()
{
int i = 0; i++;
static int j = 0;       j++;
cout << "i = " << i << " j = " << j << "\n";
}
void main (void)
```

```
{
f(); f();
}
```

Підсумовуючи, можна констатувати, що в мові C++ модифікатор static має двояке значення. Перше з них означає локальність — це стосується виключно поняття локальної статичної змінної.

Інше значення — це створення визначеного об'єкта по фіксованій адресі із локалізацією області його використання (обмеженням області видимості його ідентифікатора). Цим забезпечується існування об'єкта з момента його визначення до кінця виконання програми. Але при цьому не завжди існує можливість прямого доступу до такого об'єкта. Прикладом цього є створення статичної змінної в тілі функції чи статичного члена класу.

Клас пам'яті register використовується для створення так званих регістрових змінних, тобто таких змінних, значення яких зберігаються безпосередньо в регістрах процесора. Це, очевидно, зменшує час звертання до таких змінних і робить дуже вигідним їх використання у системах, критичних до часу виконання. Зауважимо, що описане трактування є канонічним для ранніх версій компіляторів мови C++. Вже починаючи із стандартів C89 і C99 трактування модифікатора register (яка вказівки компілятору) є дещо розширеним: "зроби доступ до змінної найшвидшим". Очевидно, що основним способом пришвидшення доступу до змінної залишається розміщення її у регістрах.

У залежності від конкретної реалізації компілятора і обчислювального середовища регістрові змінні обробляються по різному. Іноді цей модифікатор може бути повністю проігнорований. У цьому випадку він фактично виражає лише декларацію про наміри. Але у жодному випадку використанні модифікатора register не гарантує розміщення змінної саме в регістрах. Компілятор має право цей модифікатор проігнорувати, оскільки на можливість розміщення змінної в регістрі впливає багато факторів, наприклад наявність вільного регістра, розмір змінної та ін.

Початково цей модифікатор взагалі дозволялось використовувати лише стосовно типів char і int. У нинішніх реалізація мови C++ регістровий модифікатор можна застосовувати для змінних будь-якого типу. Проте великі об'єкти у регістрах фізично не можуть бути розміщені, але можуть отримати більш вищий пріоритет обробки.

При використанні модифікатора register треба дотримуватись двох правил. За першим з них треба пам'ятати, що оголошення register повинно бути строго локальним. У зв'язку з цим іноді дію модифікатора трактують як засіб економії пам'яті.

За другим правилом у стандарті C до регістрової змінної забороняється застосовувати операцію взяття адреси &, незалежно від того чи була розміщена

змінна в регістрах процесора чи ні. Стандарт C++, на відміну від ANSI C, дозволяє операцію взяття адреси стосовно регістрових змінних.

Значення регістрової змінної обробляється інструкціями в машинному коді. Тому дуже ефективним є використання модифікатора register при оголошенні керуючих змінних особливо у циклічних структурах, наприклад

```
register int f = 5;
for (register int i = 4; i > 0; i--) f = f * 1;
```

Регістрові змінні, як аргументи, функції передаються через стек. Всередині тіла функції, якщо існує можливість, вони копіюються в регістри процесора.

Наостанок підсумуємо — переведення змінних, незалежно від їх розміру, у категорію регістрових є одним із способів оптимізації програмного коду по швидкості виконання. Але у цьому випадку треба пам'ятати, що після виконання оптимізаційних процедур фізично змінна, якщо вона була розміщена у регістрах, не існує у пам'яті. Це, у свою чергу, може спричинити проблеми для сумісного використання цієї змінної, наприклад у синхронізаційних процесах.

Модифікатор const оголошує константу. Тобто таку змінну, яка негайно ініціалізується при оголошенні, і забороняється зміна її значення. У мові C++, яка є наступницею C, існує два способи оголошення констант.

Перший з них (класичний спосіб мови C) передбачає використання такої дерективи препроцесора як макрос підстановки #define. Але константа, яка визначається макропідстановкою, є типу, який визначається неявно, і не може трактуватись в контексті фізичного суб'єкта. Це означає, що наступний код є неправильним

```
#define m 4;
const int * p = &m; // неправильний код
```

Другий спосіб полягає у використанні модифікатора const. На противагу від застосування першого способу. У випадку використання модифікатора const об'єкт розглядається як фізична сутність. Тому йому явно вказується тип (для визначення розміру пам'яті й інтерфейсу доступу до неї), і явно вказується ініціалізатор — ініціалізація здійснюється оператором присвоєння. Наприклад

```
const int m  = 4;    або    int const m  = 4;
const int * p = &m;                  // правильний код
```

Важливим є те, що модифікатор const не впливає на тип змінної (див. Прикл. № 6.4), який вказується явно.

Константа обов'язково ініціалізується при оголошенні. Це означає, що код const int m; є неправильним.

Ініціалізація може здійснюватися викликом функції:

```
int f()
{
return 4;
}
...
const int i = f();
```

У ранніх версіях мови C++ модифікатор const можна було використовувати лише стосовно глобальних змінних. Тепер допускається оголошувати локальні змінні константними і навіть повертати їх значення чи адресу66 у функціях та методах класів, наприклад

```
const int* f1(void)
{
const int k = 1;
return &k;
}
int f2(void)
{
const int k = 2;
        return k;
}
...
using namespace std;
cout << *f1() << ", " << *f2() << endl;
```

Для оголошення константи, доступної в інших модулях програми використовується декларація extern поза межами будь-якої функції (наприклад, extern const i = 5;).

Ця ж декларація використовується і для організації доступу до константи ініціалізованої в іншому файлі(наприклад, extern const i;). При цьому можна змінити дію модифікатора static, наприклад,

Приклад № 6.4.

Файл var.cpp

```
#include "var.h"
extern const int i = 10; static const int j        = 20;
```

Файл - var.h

```
#ifndef VARH
#define VARH
```

```
extern const int i; extern const int j;
#endif

Файл - Ex_1_26_MSVS2012.cpp
#include <iostream>
#include <typeinfo>
#include "var.h" using namespace std;
void main (void)
{
cout << " i = " << i << ", j = " << j << endl;
cout << "Type of i is: " << typeid(i).name() << endl;
}
```

У цій програмі варто звернути увагу, на те, що у в компільованому коді модифікатор const жодним чином не відображається. Це свідчить на користь розуміння цього модифікатора виключно засобом етапу розробки.

Адресу від константної змінної можна взяти, але присвоєна вона може бути лише вказівнику на константний об'єкт, наприклад

```
const int i = 10;
int* p0 = &i;          // код неправильний
const int* p1 = &i;          // код правильний
int* const p2 = &i;          // код неправильний
```

Цим гарантується незмінність констатного об'єкта, навіть при непрямій адресації.

Одним із найчастіших випадків використання модифікатора const є вилучення літеральних констант із коду та їх заміна константної змінною, наприклад

Замість коду

```
int i = 0;
for (; i < 256; i++)... ;
i = 256 / 2;
```

рекомендується такий код

```
int i = 0;
const int size = 256; for (; i < size; i++)... ; i = size / 2;
```

Використання константного об'єкта замість літерального значення надає такі переваги:

- літеральне значення є мало інформативним (наприклад, що означає ціле число 256). У той час ідентифікатор може надати додаткову інформацію, яка може полегшити читання і розуміння коду (наприклад, ім'я size означає границю масиву);
- існує можливість безпомилкової глобальної заміни значення у всіх місцях літерального входження. Це означає, що достатньо змінити один раз значення константи і заміна буде задіяна у всіх місцях використання цієї константи. Більше зникає загроза потенційного невірної заміни або пропуску самої заміни у коді;
- використання саме константного об'єкта гарантує незмінність значення. Це означає, що зникає загроза випадкової чи помилкою модифікації значення і, відповідно, небажаної зміни поведінки програмного коду на етапі виконання програми.

Основним аргументом проти використання константних об'єктів донедавна було те, що такий об'єкт повинен бути виключно глобальним. З появою можливості оголошувати локальні константи цей аргумент був знівельований.

Наостанок відмітимо, що у C++ існує поняття константного виразу, яке отримало подальший розвиток у C++11.

Модифікатор volatile визначає так звані змінні сумісного доступу і, фактично, виступає протилежністю до регістрового модифікатора.

У явній формі цей модифікатор поміж інших дій вказує компілятору на те, що змінна не повинна зберігатись в регістровій пам'яті. При цьому забороняється будь-яка оптимізація пов'язана з volatile-змінною.

Оголошується змінна класу volatile так

```
volatile int a;        // старий формат int volatile a;
```

Інша назва volatile змінних — нестійкі змінні. Якщо об'єкт є екземпляром комплексного типу (наприклад структури), то, в додаток до самої змінної, усі її складові (наприклад поля структури) вважаються нестійкими змінними. Пояснимо це на прикладі. Нехай має місце оголошення

```
volatile struct A
{
int i;
char c;
} a;
```

тоді volatile змінними вважаються a, a.i та a.c. Ідентифікатор A не має класу пам'яті volatile, оскільки є іменем типу.

Компілятори C++ вважають, що змінні, які ніколи не зустрічаються в лівій частині оператора присвоєння, є константами і пробують оптимізувати

програму, змінюючи при цьому порядок обчислень. Більше того, з метою тієї ж оптимізації компілятор може перенести таку змінну в область швидкого доступу, наприклад в регістри.

Модифікатор volatile забороняє такі зміни. За його використання вважається, що значення змінної може бути зміненим без використання оператора присвоєння. Наприклад, у наступному коді значення змінної exit буде зчитуватись кожного разу, коли здійснюватиметься перевірка на вихід з циклу.

```
volatile bool exit = true; while ( exit ) {};
```

Це означає, що будь-які зміни будуть відразу відображатись на роботі циклу.

Подібне зчитування не буде здійснюватися у випадку звичайної змінної.

Наведений приклад ілюструє негативний вплив модифікатора volatile на швидкість виконання програми.

Модифікатори const і volatile можна використовувати одночасно. Наприклад, допустимо, що програма отримує зовні (зокрема через порт) деяке значення. Тоді це значення можна захистити модифікатором const. Це не відміняє можливості її зміни зовнішніми засобами, але захищає від внутрішніх

```
const volatile int a = (const volatile int)10;
...
a = 11;                 // заборонена операція
```

У бібліотеці <type_traits> є два шаблони template< class T > struct is_const і template< class T > struct is_volatile, які визначають наявність модифікаторів const і volatile, наприклад

```
include <iostream>
#include <type_traits>
...
std::cout << std::is_const< int >::value << '\n';
std::cout << std::is_const< const int >::value<< '\n';
std::cout << std::is_volatile< int >::value << '\n';
std::cout << std::is_volatile< volatile int >::value << '\n';
...
```

6.3. Поняття ініціалізатора

Поняття ініціалізатора є одним із логічних понять, яке виражає присвоєння значень змінним на етапі їх створення. Ініціалізатор передує оператору присвоєння і складається із значення, виразу або списку значень, який типово визначається фігурними дужками.

Ініціалізатор може бути явний або неявний. У випадку неявного ініціалізатора, для початкової ініціалізації вибираються "нульові" значення. Тут треба зауважити, що трактування "нульових" значень є залежним від типу. Наприклад, для змінної цілого типу — це ціле число 0, для вказівника — значення константи NUUL; для об'єкта — це конструктор по замовчуванні, тощо. У випадку явного ініціалізатора значення початкової ініціалізації вказуються у вихідному коді.

Усі вирази, які використовуються у явному ініціаліалізаторі незалежно від класу пам'яті static чи extern повинні бути:

- константними виразами (літеральними константами, або виразами, обчислення яких повертає константу або lvalue);
- виразами, які зводяться до адрес вже існуючих глобальних змінних. Для останніх допускається зміщення на константу.

Наведемо приклади правильних явних ініціалізаторів

```
const int v = 1; // ініціалізація у форматі присвоєння
int z1;
int z2 = (int)tan( (double)1.0 ); int z3 = v;
int* z4 = &z2;
int* z5 = new int[z3]; int* z6 = new int + 2;
int z7(1); // ініціалізація у форматі конструктора
```

Змінні класу auto та register і їхні адреси не можуть використовуватись приініціалізації глобальних змінних. Ініціалізація таких змінних відбувається кожного разу при вході в програмний блок. Але лише у випадку виключно явної ініціалізації.

Явна ініціалізація може бути реалізована засобом списків ініціалізації. Наприклад

```
int z2 {1};
```

Ініціалізація за допомогою списків забороняє звуження (анг. narrowing) типів. Суть цієї заборони ілюструє такий приклад

```
int z1 = 7.3;
int z2 {7.3};
```

Змінна z1 в процесі ініціалізації отримає значення 7, тобто обрізане від типу double значення. Напротивагу їх змінна z2, для ініціалізації якої використано список, значення не отримає. Буде згенеровано помилку періоду компіляції, оскільки списки звуження типу забороняють.

Формат ініціалізації за допомогою списків більше призначений для ініціалізації змінних складених типів. Відповідно виклад кожного складеного типу передбачатиме розгляд ініціалізації за допомогою списків.

Неявної ініціалізації для змінних класу auto та register немає. Тому за відсутності явної ініціалізації початковим значенням для них буде виступати вміст комірки пам'яті, який залишився після попередніх операцій з пам'яттю.

Для випадків змінних складених типів та класів поняття ініціалізатора буде розглядатись додатково.

Описане поняття ініціалізатора дуже спрощено описує процедуру початкового присвоєння значення. Стандарт C++11 визначає підсистему методів ініціалізація.

6.4. Життєвий цикл змінної

Кожна змінна, як фізичний об'єкт (або фізична сутність), в процесах розробки повинна розглядатись в контексті свого життєвого циклу, який складається із чотирьох етапів: неіснування, часткової побудови, створення екземпляру (інстанаціювання) та часткового знищення. Вочевидь, що ця схема є загальною і приймається для випадків об'єктів, які володіють власними алокаторами, ініціалізаторами та деалокаторами. У випадку простих змінних перелік етапів та й самі етапи є трохи спрощеними. Але на рівні розуміння логістики змінної і маніпуляцій з нею треба розглядати повний життєвий цикл, тобто такий, у якому розрізняються усі етапи. У будь-якому випадку виділення фізичної пам'яті повинно бути здійснено до побудови об'єкта, а її звільнення – після його знищення.

Створення будь-якого об'єкта (змінної) визначає його ОД і є можливим завдяки процесу виділення пам'яті. Тут розглядають такі випадки:

- якщо пам'ять виділяється компілятором (або лінкувальником) у адресному просторі процесу (глобальному адресному просторі), то розглядаються глобальні об'єкти, тобто такі, які створюються за межами будь-якої функції. Це стосується також функції main(). Звільнення цієї пам'яті відбувається після завершення цієї функції.
- якщо пам'ять виділяється компілятором (або лінкувальником) шляхом налаштування вказівника стеку (підобласті в глобальному адресному просторі), то розглядаються об'єкти стеку, тобто такі, які існують обмежений час (час, який визначається виконанням функції). За формальним підходом вони створюються у точках своїх оголошень і автоматично знищуються, коли управління виходить за область їх видимості.
- якщо пам'ять виділяється користувацькими інструкціями (оператори new) під час виконання коду у глобальній (чи локальні) купі (підобласті глобального адресного простору), то розглядаються динамічні об'єкти (heap objects). Звільнення цієї пам'яті також здійснюється користувацькими інструкціями (оператори delete). Стандарт мови гарантує коректні виклики

конструкторів та деструкторів у взаємозв'язку з процедурами виділення та звільнення пам'яті.

Об'єкт (змінна) можуть бути частиною іншого об'єкта. У такому випадку їх життєвий цикл і, відповідні, процеси створення та знищення визначаються життєвим циклом і відповідними процесами іншого (складеного) об'єкта.

Лекція 7. Переліки. Структури. Розміщення структур у пам’яті.

До категорії складених типів у стандарті C++11 відносять: масиви, функції, вказівники (на будь-якого виду), посилання, структури (класи) й об'єднання та шаблони. Підтримуються модифікатори const і volatile.

Складені як композиція простих є результатом процесу побудови типу самим користувачем. Звідси походить друга назва користувацькі типи. Зазвичай користувацькими вважаються розробки класів (структур, об'єднань) та шаблонів. Це пояснюється тим, що на їх розробку витрачається набагато більше часу, а ніж на організацію масиву чи переліку. Але ця думка є хибною, і до будь-якого складеного типу треба підходити з точки зору його розробки користувачем.

Для складених типів характерним є мінімальна кількість операцій, яка визначена стандартом мови і реалізована виробником компілятора. До цих операцій відносять алокацію, організацію доступу та порозрядне копіювання (для структур, класів і об'єднань). Відповідно розробка таких типів як класи (структури), об'єднання та шаблони, у першу чергу потребує додаткової розробки операцій над змінними цих типів. Це вимагає набагато більше часу розробки у порівнянні із побудовою масивів, переліків чи оголошення вказівників. У зв'язку з цим класи (структури), об'єднання та шаблони називають користувацькими типами. Хоча це зовсім вірною. Оскільки складені типи є композицією простих, визначення будь-якого складеного типу є креативною дією користувача. А це визначає користувацьким будь-який тип з категорії складених.

Бібліотека <type_traits> надає шаблон template< class T> struct is_compound для визначення складених типів, наприклад

```
#include <iostream>
#include <type_traits>
void main (void)
{
class A {};
std::cout << (std::is_compound< A >::value
                                    ? "T is compound"
                                    : "T is not a compound") << '\n';
 std::cout << (std::is_compound< int >::value
                                    ? "T is compound"
                                    : "T is not a compound") << '\n';
}
```

7.1. Переліки

Концепція перелікового (перерахункового) типу є широко вживаною, хоча і не обов'язковою. Формально, цей тип є набором (зазвичай малим) іменованих цілих констант. Наприклад, оголошення

```
enum days { mon, tue, wed, thu, fri, sat, sun };
```

визначає перерахунковий тип days. Як видно з прикладу, означення перерахункового типу розпочинається із зарезервованого слова enum, імені типу і списку імен допустимих значень в круглих дужках.

Змінні цього типу можуть приймати виключно сім значень, які, за допомогою імен, перелічені в круглих дужках. Усі допустимі значення перерахункового типу мають власні імена (так, наприклад, серед усіх можливих значень типу bool є тільки два і вони уособлюються ідентифікаторами: true і false), наприклад

```
days day;
day = wed;
```

Якщо відразу створюються змінні даного типу, то ім'я може бути опущено. Такий перерахунковий тип називається анонімним. Анонімний перерахунковий тип не може надалі використовуватись для створення змінних, оскільки відсутнім є ідентифікатор типу, наприклад:

```
enum { spades, hearts, diamonds, clubs } card1, card2;
...
// присвоєння значень змінним перерахункового типу
card1 = spades;
card2 = card1;       // оператор присвоєння для змінних
...
if ( card2 == clubs ) { ... }
```

У наведеному прикладі, створено дві змінні анонімного перерахункового типу, які уособлюють картки замовлення. Неможливо створити ще декілька карток, оскільки перерахунковий тип є анонімним.

Анонімні перерахунки іноді вводять з метою впровадження в програму іменованих констант для обмеженого використання і без застосування модифікатора const.

З внутрішньої сторони, константи перерахункового типу, наприклад типу days, є зв'язаними із відповідними цілими значеннями, значення яких розпочинаються з нуля. У наведеному прикладі оголошення типу days значення mon буде трактуватись як ціле число 0, значення tue як 1 і т.д.

Допускається деталізація присвоєння цілих значень. Саме цими значеннями будуть ініціалізовуватись константи перерахункового типу, наприклад:

```
enum days { mon, tue = 0, wed = 0, thu = 0, fri = 0, sat, sun };
```

Правила визначення значень є такими:

- перший елемент буде завжди мати значення нуль, якщо не було присвоєно жодного іншого значення. У наведеному прикладі, mon буде зв'язаний із 0;
- будь-який інший елемент, якщо він не конкретизувався значенням, буде зв'язаний із значенням більшим на одиницю від значення попередньої константи. У наведеному прикладі, усі константи від tue до fri будуть відповідати цілому значенню 0. А от константа sat буде мати значення на одиницю більшу від значення константи fri і рівна 1. Подібно, константа sun буде мати значення 2.

Значення перерахункових констант повинні завжди бути у зростаючому порядку, але не обов'язково з різницею в одиницю (вони можуть мати також від'ємні значення). Наприклад:

```
enum days { mon, tue = 0, ..., fri = 0, sat, sun = 3 };
```

У цьому визначенні sat буде рівним 1, а константа sun – 3.

Наведемо приклад, який узагальнює усе сказане вище.

Приклад № 7.1.

```
#include <iostream>
#include <string>
using namespace std;
enum days { mon, tue = 0, wed = 0, thu = 0, fri = 0, sat, sun };
void info(days day)

{
static string  dayType[] =
{"       weekday", " saturday", "  holiday"};
int rate = 100*(1 + day);
cout << "Type of day:"     << dayType[day]    << ". "
<< "The rate is: " << rate << " USD" << endl;
}
void main (void)
{
info(mon);
days day = sat; info(day);
```

```
day = sun; info(day);
}
```

Прокоментуємо детально наведену програму. У ній оголошено перерахунковий тип days. Це оголошення є глобальним (поза межами функцій), що визначає глобальний простір як ОВ для ідентифікатор days. Тобто він є видимим (доступним) у довільному місці буд де у файлі від моменту оголошення і до кінця файлу. В тому числі і в тілах інших функцій, зокрема у тілі функції main().

Наступним оголошується функція info(). Її параметром є змінна типу days. У тілі функції main() є декілька викликів функції info(). У першому виклику їй аргументом передається константа mon – окрема літеральна константа означена в days. У решти викликах – змінна day, яка є типу days, але ініціалізована у кожному випадку окремим значенням.

З наведеного коду видно, що створення змінної day нічим не відрізняється від створення змінних інших типів: першим слідує тип, а далі ім'я змінної. Опційно, в даному рядку здійснено ще й початкову ініціалізацію цієї змінної допустимим значенням sat. При цьому важливим є те, що аналогічне за своїм результатом присвоєння змінній day значення цілого типу

```
days day = 1; // недопустиме
```

є неправильним. Незважаючи на те, що значення типу days представляються цілими числами, наведене присвоєння у C++ є забороненим. Виходом з цієї ситуації є явне використання оператора приведення типу

```
days day = (days)1; // допустиме
```

У тілі функції info(), визначено масив стрінгів розмірністю три елементи. Основним завданням функції є вивід типу інформації про деякий коефіцієнт, який є залежним від типу дня.

Для індексації елементів масиву (як значення типу int) використовується змінна типу days: dayType[day]. Це є правомірно, оскільки в цьому випадку відбудеться автоматичне неявне приведення змінної day до типу int, а результат використовуватиметься індексом.

Варто звернути увагу на дію '1 + day'. Перший операнд є типу int, а другий типу days. Подібно до попереднього випадку day буде неявно приведено до типу int. А результат приведення буде виступати другим операндом для виконання дії присвоєння.

Ще раз нагадаємо, що зворотне перетворення - з типу int у тип days – є забороненим. Це означає, що дія ' day + 1' є недопустимою. А виклик функції info(k) з аргументом k типу int є забороненим (навіть у випадках, коли б k було

рівне 0, 1 або 2). Будь-який аргумент, який передається функції info() повинен бути типу days. Такий підхід гарантує, що функція info() буде завжди викликана з правильним аргументом: змінною типу days, яка відповідає значенням 0, 1 або 2. Це правило дозволяє гарантувати валідність індексу (вихід за допустимі межі) всередині функції.

У загальному випадку компілятори здійснюють перевірку типів аргументів, які передаються у функцію: перерахункові типи дозволяють зробити перевірку не тільки за типами, а й за валідністю аргументів.

Якщо екземпляр перерахункового типу є глобальною змінною, то по замовчуванні вона буде ініціалізуватись першим значення, яке зв'язане числом 0. Наприклад

Приклад № 7.2.

```
#include <iostream>
using namespace std;
enum days0 { mon0, tue0 = 0, wed0 = 0} day0;
enum days1 { mon1, tue1 = 0, wed1 = 0} day1;
enum days2 { mon2 = -1, tue2, wed2 = 1} day2;
enum days3 { mon3 = -1, tue3 = 1, wed3 = 1} day3;
void main (void)
{
cout << (( day0 == mon0 ) ? "Is initialized - ":
"Didn't initialized - ") << "day = "<< (int)day0 << endl
<< (( day0 == tue0 ) ? "Is initialized - ":
"Didn't initialized - ") << "day = "<< (int)day0 << endl
<< (( day1 == mon1 ) ? "Is initialized - ":
"Didn't initialized - ") << "day = "<< (int)day1 << endl
<< (( day1 == tue1 ) ? "Is initialized - ":
"Didn't initialized - ") << "day = "<< (int)day1 << endl
<< (( day2 == mon2 ) ? "Is initialized - ":
"Didn't initialized - ") << "day = "<< (int)day2 << endl
<< (( day3 == mon3 ) ? "Is initialized - ":
"Didn't initialized - ") << "day = "<< (int)day3 << endl;
}
```

Глобальна змінна day0 ініціалізується нульовим цілим значенням. Таке значення відповідає константі mon0.

Ініціалізується нульовим цілим значенням змінної day1 призводить до того, що така змінна одночасно рівна константам mon1 і tue1.

У випадку змінної day2 ініціалізація по замовчуванні призведе до рівності константі tue2. І нарешті у випадку змінної day3, ініціалізація нульовим

значенням призведе до того, що ця змінна не буде містити допустимого значення і формально вважається неініціалізованою.

Основні проблеми переліків, які існувала ще у мові C, полягали у наступному:

- на практиці переліки представляються цілими числами, що дозволяє здійснювати порівняння між двома значеннями із різних переліків. Формальним захистом від цього була заборона неявного перетворення цілих чисел, або елементів одного переліку в інший перелік;
- спосіб представлення в пам'яті є залежним від реалізації, а тому виникає проблема переносимості
- елементи переліку входять до однієї ОД, що забороняє створювати елементи з однаковими ідентифікаторами у різних переліках.

Для вирішення цих проблем у мові C++11 перелік розглядається в контексті простору імен. Такий підхід реалізовується введенням нового виду переліку так званого типобезпечного переліку (ТБП). Він визначається оголошенням enum class (або enum struct), наприклад

```
enum class days
{ mon, tue = 0, wed = 0, thu = 0, fri = 0, sat, sun };
```

ОД елементів ТБП визначається ОВ ідентифікатора переліку. Доступ до елементів визначається обов'язковим використанням ОДД із іменем типу, наприклад days::mon. Забороненим є неявне перетворення у цілі числа, наприклад недопустимим є код int x = days::mon;.

Для звичайних переліків базовий тип є невизначеним. У C++11 ТБП неявно будується на основі типу int – базовий тип. Проте базовий тип для ТБП, а також для звичайного переліку у новому стандарті може бути заданий явно, наприклад.

```
enum class days1 : unsigned long
{ mon, tue = 0, wed = 0, thu = 0, fri = 0, sat, sun };
enum days2 : unsigned long
{ mon, tue = 0, wed = 0, thu = 0, fri = 0, sat, sun };
```

У наведеному коді елементи звичайного переліку days2 стандарт C++11 розглядає визначеними просторі імен days2. Але для забезпечення сумісності із звичайними переліками ці елементи є видимими в глобальній ОВ. Тому для організації доступу до них явне використання ОДД є необов'язковим. У цьому полягає розширення поняття переліку C++ у мові C++11.

Зазначимо, що явне задання базового типу з поміж іншого нівелює проблему переносимості.

C++11 дозволяє попереднє оголошення будь-яких переліків. При цьому на них накладається обмеження – розмір переліку повинен бути явним чи неявним способом визначений, наприклад

```
enum days1 ;                        // помилка - невизначений базовий тип
enum days2 : unsigned long;         // правильно
enum class days3 : unsigned long;   // правильно
enum class days4 ;                  // правильно неявно базовим типом є
int
```

Для перевірки перерахункового типу бібліотека <type_traits> має шаблон template< class T > struct is_enum, наприклад

```
#include <iostream>
#include <type_traits>
class A {};
enum E {};
enum class Ec : int {}; // не підтримується у MS VS 2010
void main(void)
{
std::cout << std::boolalpha;
std::cout << std::is_enum< A >::value << '\n';
std::cout << std::is_enum< E >::value << '\n';
std::cout << std::is_enum< Ec >::value << '\n';
std::cout << std::is_enum< int >::value << '\n';
}
```

7.2. Структури

Структури мови C++, подібно до C, є типами, які являють собою іменовану сукупність компонентів різних типів. Ці компоненти називаються полями, або елементами структури.

Елементами структури може бути:

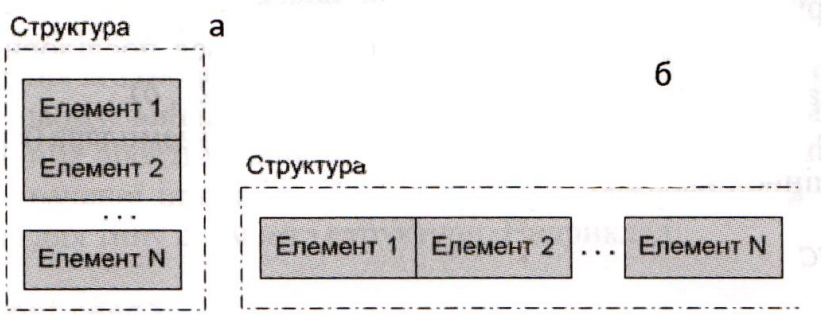
- змінна будь-якого допустимого типу;
- бітове поле;
- функція.

Оголошення структури є таким:

```
struct [ім'я_структури]
{
тип    ім'я_ елемента1;
тип    ім'я_ елемента2;
...
тип    ім'я_ елементаN;
} [список_змінних];
```

Тут у квадратних дужках наведено необов'язкові сутності конструкції.

На Рис. 7.1 наведено схемне представлення структури її елементами, зокрема на Рис. 7.1а – логічна схема структури, а на Рис. 7.1б – схема фізичного розміщення у пам'яті.



а) логічне б) фізичне

Рис. 7.1. Представлення структури

Типом елемента структури може бути довільний тип, у тому числі структурний, зокрема це може бути:

- інша структура:
- вказівник на даний структурний тип;
- будь-який інший тип, який рекурсивно не посилається на дану структуру. Наприклад

```
// Попереднє оголошення структури
struct S2;
// Визначення структури і вкладеної структури
struct S1
{
char a[2];
struct S3 {} *ps; // вкладена структура
} z;
// Повне визначення попередньо оголошеної структури
struct S2
{
char a[2];
S2 strc;        // Помилка -        на саму структуру
                // може посилатись тільки вказівник
};
```

Структура не може містити у якості вкладеної структури саму себе, але може містити елемент, який є вказівником на саму себе, наприклад

```
struct S
```

```
{
S *p;  // поле - вказівник на структуру
} s;     // створення екземпляра структури
```

Оголошення структури є визначенням користувацького типу. При цьому:

- Розмір типу структура рівний сумі розмірів усіх її полів. Хоча можуть бути випадки, коли загальна пам'ять, яка виділяється під структуру, перевищує суму полів.
- Пам'ять під усі елементи структури виділяється послідовно для кожного елементу в порядку їх оголошення.
- Фізично пам'ять виділяється в процесі створення екземпляра (копії) структури, тобто змінної, наприклад

```
struct STRUC
{
int     i;
float   f;
char   s[300];
};
...
STRUC strc;  // створення екземпляра структури і
// визначення розміру
std::cout << "Struct size: " << sizeof(strc) << std::endl;
...
```

У наведеному коді список_змінних був опущеним. У загальному випадку він може містити імена змінних, ідентифікаторів масивів та вказівники.

Елементи структури можуть мати модифікатори доступу: public, private і protected.

По замовчуванні усі елементи структури оголошуються відкритими, тобто із модифікатором public. Для звертання окремих елементів структури використовуються оператори доступу operator . і operator ->82, які повертають посилання на адресу пам’яті, виділену під цей елемент. Наприклад наведемо приклад доступу до полів STRUC з її екземпляра strc

```
std::cout << "Field sizes:" << sizeof(strc.i) << ", " <<
sizeof(strc.f) << ", " << 300*sizeof(char) << ". ";
```

Екземпляр структури, наприклад strc, треба розуміти визначеним інтерфейсом, який, оскільки містить адресу, надає доступ до деякої ділянки пам'яті розміром: sizeof(int) + sizeof(float) + sizeof(300* char). Усередині цієї області поля структури відіграють роль інтерфейсів доступу до окремих підділянок. Якщо на ділянку змінної strc накласти інший інтерфейс, то її вміст

можна прочитати у іншому трактуванні. Це означає, що структура, насамперед визначає достатньо гнучкий інтерфейс доступу до вмісту ділянок пам'яті адресного простору. І ще важливим є те, що цей інтерфейс може бути користувачем змінений. Початкова ініціалізація полів структури може бути відсутня або забезпечується:

- конструкторами.
- механізмом по замовчуванні.
- явно заданими ініціалізаторами (списками ініціалізації) .

Технологія конструкторів, ініціалізація за їх допомогою та її розвиток у C++11 (наприклад універсальна ініціалізація) розглядатимуться у другому розділі.

Ініціалізація по замовчуванні стосується виключно глобальних екземплярів структури. У цьому випадку їх поля ініціалізуються нульовими значеннями, подібно до змінних простих типів. Якщо поле структури є складеним типом, - то за відсутності визначеної ініціалізації конструктором, запускається ініціалізація по замовчуванні для типу елемента структури. Наприклад

```
#include <iostream> struct A { int i; }; struct B
{
A a; int i;
} b;
void main(void)
{
std::cout << b.i << b.a.i ;
}
```

У випадку використання ініціалізатора (списків ініціалізації), останній визначається у фігурних дужках, а значення відповідних полів розділяються комою і задаються у послідовності оголошення полів: від першого елемента до останнього. Не можна передати ініціалізаційне значення, наприклад, останньому елементу не передавши явно значень для усіх попередніх елементів. Значення для кожного поля задається ініціалізатором його типу. Наприклад

Приклад № 7.3.

```
#include <iostream>
#include <string>
using namespace std; struct A
{
int i;
char* s; int m[3];
// int m[]; - розмір неможливо визначити за ініціалізатором
// і він буде нульовим
```

```
} a = {1, "aaaa",{1,2,3}};   // явна ініціалізація усіх полів
A b = {2};      // явна ініціалізація тільки одного поля
A c = {3,"c",{-1}};   // явна ініціалізація двох полів і
// 0-го елемента третього
//A d = {, "bbbb"};  // помилка ініціалізації
void main(void)
{
cout << "a.i = " << a.i << " Str Len: " << ( (a.s) ? strlen(a.s)
: 0) << " Mas Count:" << sizeof(a.m)/sizeof(int) << " Zero index el.:" << a.m[0] <<
endl
<< "b.i = " << b.i << " Str Len: " << ( (b.s) ? strlen(b.s)
: 0) << " Mas Count:" << sizeof(b.m)/sizeof(int) << " Zero index el.:" << b.m[0] <<
endl
<< "c.i = " << c.i << " Str Len: " << ( (c.s) ? strlen(c.s)
: 0) << " Mas Count:" << sizeof(c.m)/sizeof(int) << " Zero index el.:" << c.m[0] <<
endl;
}
```

Відсутність явної ініціалізації полів s і m змінної b призвела до їх ініціалізації по замовчуванні. У результаті цього поле s буде рівним NULL, а елементи масиву m з індексами 1 і 2 – рівними 0.

У ініціалізаторі можна використовувати ідентифікатори змінних, наприклад створення екземплярів структури A може бути таким

```
A a1 = {i: 1, s: "aaaa",m: {1,2,3} };
A a2 = {.i = 1, .s = "aaaa",.m = {1,2,3} };
```

У випадку масивів структур, або структур, поля яких є, у свою чергу, також структурами, допускається використання списків ініціалізації. Наприклад

```
A a3[] = { {1, "aaaa", {1,2,3} }, {2, "bbbb", {3,4,5} } };
```

Ініціалізація списком допускається навіть при передаванні об'єкта у тіло функції, наприклад

```
void f(A a)
{ ... }
...
f( {0, "aaa", {1, 2, 3} } );
```

Використання списків ініціалізації дозволяється лише стосовно структур, які є POD-типами. Якщо структура не є POD-типом то списки ініціалізації є

забороненими. Це обмеження призвело до появи у мові C++11 шаблонного класу std::initializer_list, який поширює концепцію списків ініціалізації на POD-типи.

Для структур по замовчуванні визначено оператор присвоєння у форматі функції порозрядного копіювання. Це означає, що валідною є операція присвоєння, наприклад a = b, яку можна записати з використанням функції void* memcpy(void*, const void*, size_t) (визначена у ФЗ – <memory.h>), наприклад memcpy(&a, &b, sizeof(b)).

У випадку відсутності ідентифікатора структури створюється анонімна структура (АС). Екземпляр такої структури треба відразу створювати, оскільки за відсутності ідентифікатора, неможливо буде явно85 створити екземпляр АС, наприклад

```
struct
{
int x;
} s1, s2[2], *s3; // створено 3 екземпляри структури
```

Анонімним може бути поле структури, наприклад

```
struct A
{
int ; int x;
} s;
```

Таке поле дуже часто використовується для:

- вирівнювання області пам'яті;
- непрямого доступу до поля структури. Наприклад у наведеному прикладі пам'ять, розміром 2*sizeof(int), під екземпляр s буде виділена. А це означає, що доступитись до неї, зокрема до першої ділянки розміром sizeof(int), можна, правда не засобом змінної s, оскільки поле структури є анонімним.

Із визначенням структури можна використовувати оператор typedef, наприклад

```
typedef struct STRUCT
{
int i;
} st;    // псевдонім іменованої структури

typedef STRUCT STRC;     // псевдонім іменованої структури
typedef st STRC2;  // псевдонім від псевдоніма

typedef struct                // псевдонім анонімної структури
```

```
{
int i;
} stuc;

stuc a0;
st a1;
STRC a2;
STRC2 a3;
STRUCT a4;
```

У мові C++ визначення структури (та й об'єднання) було розширено таким чином, щоб у їх оголошення стало можливим включення функцій-членів, а також конструкторів та деструкторів. Це зробило структуру (та й об'єднання) синтаксично схожими на клас.

У результаті такого розширення поняття структури дуже часто розглядають дві окремі конструкції: структури мови C та мови C++. Основними відмінностями структур C++ від структур мови C є таке:

- структура C++ є класом, тобто базисом реалізації теорії ОПП у мові. Це означає, що усі можливості, засоби та технології можуть використовуватись стосовно структур C++. Основною відмінністю класу від структур (та об'єднань) є оголошенням членів по замовчуванні: у структурі (та об'єднанні) вони відкриті, а у класі – закриті. Це означає, що зарезервоване слово class без жодних обмежень може бути замінене словом struct.
- однотипні поля можна оголошувати в одному рядку;
- імена полів структур можуть збігатись з іменами інших змінних, які оголошені поза структурою;
- після закриваючого оголошення структури дужки можна оголошувати одну чи декілька змінних екземплярів типу структура;
- оскільки оголошення структури (класу, об'єднання, переліку) є оператором, то воно повинно закінчуватись символом ';' навіть у тих випадках, коли жодного екземпляра не створювалось.

У контекстів класів в екземплярах структур C++ можна використовувати оператор ОДД, наприклад

```
...
struct S1
{
int i;
struct S2                    // вкладена структура
{
int i;
};
```

```
};
S1 s1 = {1}                  // створення екземпляра структури
S1::S2 s2 = {2};             // створення екземпляра вкладеної структури

void main(void)
{
cout << s1.S1::i<<'n';           // доступ за допомогою ОДД
cout << s1.S2::i<<'n';           // доступ за допомогою ОДД
//cout<<s1.S1::S2::i<<'n';       // доступ до неіснуючого поля
}
```

Як вже відзначалось розмір структури визначається сумою розмірів її полів (насправді сумою розмірів типів полів). Але, на практиці, ця теза не завжди справджується. Розмір структури в пам'яті залежить від налаштувань компілятора, директив у коді та обчислювального середовища. Можливі випадки і дуже часті, коли розмір структури є більшим за сумарний розмір полів. Наприклад, розглянемо код для компілятора MS VS 2012 у середовищі 32-х розрядної версії ОС Windows 8 (платформа Win32)

Приклад № 7.4.

```
#include <iostream>
using namespace std;
struct S {};    // порожня структура
struct S1
{
char a[2]; char c, *s; S *p;
short i;
} z = { "a", 'b', "qwerty", new S };
void main(void)
{
cout << "Sizes:" << endl
<< "z.a = " << sizeof(z.a) << " Adress = " << hex
<< int(&z.a) << dec << endl //розмір і адресвказівника
<< "z.c = " << sizeof(z.c) << " Adress = " << hex
<< int(&z.c) << dec << endl //розмір масиву
<< "z.s = " << sizeof(z.s) << " Adress = " << hex
<< int(&z.s) << dec << endl //розмір простого типу
<< "z.p = " << sizeof(z.p) << " Adress = " << hex
<< int(&z.p) << dec << endl //розмір вказівника
<< "z.i = " << sizeof(z.i) << " Adress = " << hex
<< int(&z.i) << endl //розмір простого типу
<< dec <<"z = " << sizeof(z) << " Adress = " << hex
```

```
<< int(&z) << dec << endl//розмір екземпляра
<< "*z. = " << sizeof(*z.p) << endl //розмір об'єкта посилання
<< "S = " << sizeof(S) << endl;//розмір порожньої структури
}
```

За результатами компіляції видно, що розмір змінної z на 3 байти перевищує сумарний розмір полів структури. Чому? Відповідь на це питання відображена на Рис. 7.2.

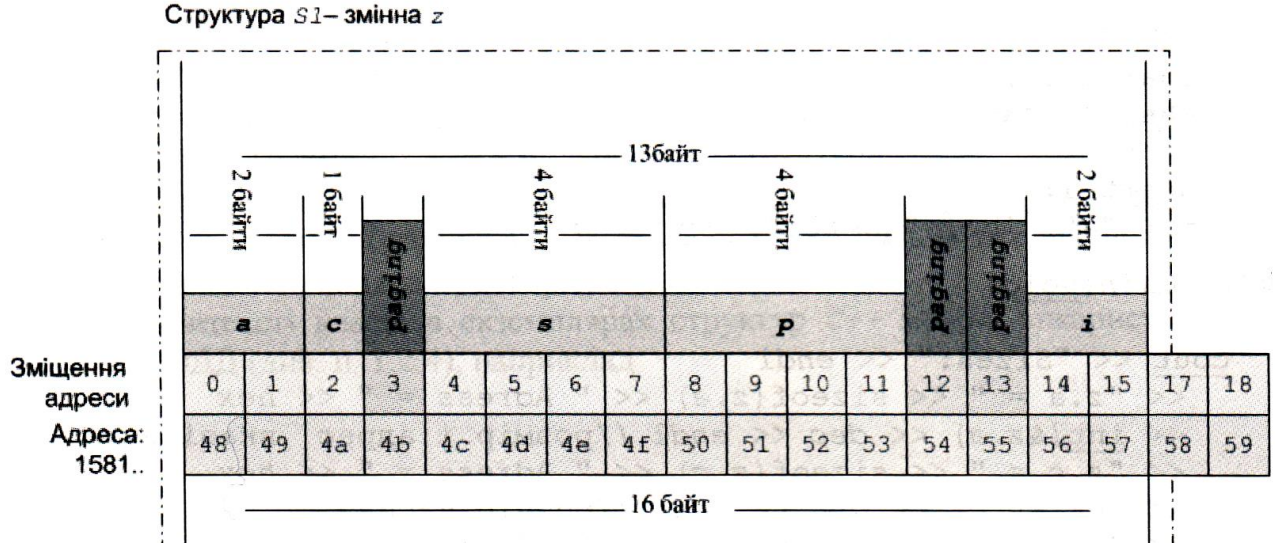


Рис. 7.2. Фізичне розміщення у пам'яті полів змінної z з 'Прикл. № 7.4

Для розуміння схеми фізичного розміщення треба знати, що для більшості сучасних апаратних архітектур існує одне дуже важливе правило: якщо процес запису/зчитування інформації довжиною N-байтів, здійснюється однією командую, то адреса початку цієї інформації повинна бути кратною N.

Для забезпечення цього правила існує процедура вирівнювання даних по довжині машинного слова (див. тлумачний словник). Суть процедури вирівнювання полягає у тому, що 1-байтові поля не вирівнюються, 2-байтові – вирівнюються по парних адресах, 4-байтові – по позиціях, які кратні чотирьом і т.д. У результаті цього при вирівнюванні полів до ділянки пам'яті, яка відводилась під змінної z, у певних позиціях було додано три невидимих байти (невидимих з точку зору інтерфейсу структури). Ці байти називаються paging bytes (на Рис. 7.2 вони позначені paging). Завдяки цьому збільшився фізичний розмір змінної z. Тепер якщо б у структуру S1 додати три байти у відповідних позиціях, наприклад

```
struct S1
{
        char a[2];
        char c, added_1, *s;
        S *p;
```

```
        char added_2, added_3;
        short i;
} z;
```

то фізичний розмір змінної z не зміниться.

Директивою #pragma pack для визначеної структури можна задати власне вирівнювання і тим самим відмінити вирівнювання по замовчуванні, наприклад

```
#pragma pack(push, 1) // вирівнювання по 1 байту
struct S1
{
char a[2];
char c, *s;
S *p;
short i;
} z = { "a", 'b', "qwerty", new S };
#pragma pack(pop) // кінець
```

Дія директиви #pragma pack стосується першої структури (чи об'єднання), яке слідує за дир тивою #pragma pack із ключем push. Детальні про цю директиву можна прочитати у довідко підсистемах компіляторів.

Відміна вирівнювання забезпечує:

- компактне розміщення фізичних сутностей пам'яті,
- можливість маніпулювання вмістом для пам'яті шляхом простих пересувань вказівників відносно адреси початку зберігання екземпляра структури.

З іншого боку, ця відміна сповільнює швидкодію, оскільки з'являється неоптимальність в операції зчитуванні/записі за одну дію.

У Прикл. № 7.4 є ще один цікавий момент, а саме це структура S і її екземпляр, який адресується вказівником z.p. Формально її розмір є нульовим, але оскільки стосовно екземпляра структури повинна бути валідною операція взяття адреси, то її розмір визначений розміром мінімальної ділянки пам'яті в 1 байт.

Для перевірки типу структура бібліотека <type_traits> має шаблон template< class T > struct is_class, наприклад

```
#include <iostream>
#include <type_traits> using namespace std; struct A {};
class B {};
enum class C {}; // не підтримується у MS VS 2010
union U {};
enum D{};
```

```
void main(void)
{
cout << boolalpha;
cout << is_class< A >::value << '\n';
cout << is_class< B >::value << '\n';
cout << is_class< C >::value << '\n';
cout << is_class< int >::value << '\n';
cout << is_class< D >::value << '\n';
cout << is_class< U >::value << '\n';
}
```

Лекція 8. Бітові поля. Об’єднання.

8.1. Бітові поля

Однією із слабких сторін ANSI C було те, що найменшим адресованим розміром було 8 біт, тобто один байт. У C++ з'явилась можливість подолати цей недолік через використання бітових полів (БП).

В основу БП покладено поняття структури. Бітове поле є різновидом елемента (поля) структури, розмір якого можна задати в бітах. Визначення бітового поля має вид

```
struct [ім'я_структури]
{
тип    ім'я_ елемента1: к-сть біт; типім'я_ елемента2: к-сть біт;
...
тип    ім'я_ елементаN: к-сть біт;
} [список_змінних];
```

Наприклад

```
struct A
{
        unsigned i:588;
        int j:3;
        int k;
};
        main()
{
        A a;
        a.i = 1; a.j = 16;
}
```

Діапазон допустимих значень БП визначається його розміром.

При створенні екземпляра типу структура з бітовими полями для останніх пам'ять виділяється починаючи з молодших розрядів і її (пам'яті) розмір є кратним байту. Наприклад, розмір структури

```
#pragma pack(push, 1)
struct S1
{
        int a:4;
         char s;
```

```
        int x;
        int b:4;
} z;
#pragma pack(pop)
```

є рівним 13 байтам, тобто маємо sizeof(int) + sizeof(char) + sizeof(int)+ sizeof(int) = 4 + 1 + 4 + 4 = 13 байт.

Розмір структури

```
#pragma pack(push, 1)
struct S2
{
int a:4; int b:4; char s; int x;
} z;
#pragma pack(pop)
```

вже є рівним 9 байтам, оскільки з'явилось два оголошення БП, які відносяться до одного фізичного поля розміру sizeof(int). У такому випадку БП a і b будуть зберігатись у одному полі довжиною sizeof(int). У перших чотирьох розрядах першого байту цього фізичного поля зберігається значення поля a, а у других чотирьох розрядах першого байту – значення поля b. Тобто у фізичному полі доступними є лише перших 8 розрядів, решта 24 розряди є недоступними.

Для того, що зробити доступними усі розряди фізичного поля і при цьому не повинен змінитись розмір структури, потрібно оголошувати такі БП (обов'язково типу фізичного поля, у нашому випадку – це int, і з неперевним послідовним оголошенням), сумарна порозрядна довжина яких повинна бути рівно 32 бітам, наприклад

```
#pragma pack(push, 1)
struct S3
{
int a:8, b:8, d:8, c:4, f:2, g:2;
int b:4;
char s;
int x;
} z;
#pragma pack(pop)
```

Із вилучення директиви #pragma pack запускається процедура вирівнювання. Відповідно розмір структури S1 буде рівний 16 байтам, а S2 і S3 – 12 байтам.

Елементи БП іменувати не обов'язково і цим самим створювати неіменовані або анонімні БП (АБП). Це дає можливість виділяти інформативні поля і ігнорувати непотрібні, наприклад попередній приклад можна записати так

```
struct A
{
        unsigned :5;
        int j:3;
};
```

Існує багато ситуацій, в яких є виправданим використання АБП. Зокрема, вони можуть використовуватись для заповнення відповідної області пам'яті. Якщо це поле є полем нульової довжини (для АБП дозволяється і таке), то можна задавати вирівнювання наступного БП по границі чергового елемента пам'яті.

До АБП неможливо звернутись по імені, його неможливо ініціалізувати, а також прочитати його значення шляхом безпосереднього звертання до БП.

БП можуть бут елементами звичайних структур, наприклад

```
struct A
{
        unsigned j:3;
};
struct B
{
        unsigned i:5;
};
struct C
{
        struct A; // не анонімне поля
        struct B; // не анонімне поля
};
```

Структура *C* має порожнє тіло. ЇЇ не треба плутати із структурою

```
struct C
{
        A a;
        B b;
};
```

розмір якої рівний 8 байтам.

БП можна використовувати в звичайній структурі, наприклад

```
struct A
{
        int i;
        unsigned j:3;
};
```

Бітові поля є машинно-залежними. А тому їх використання супроводжується певними обмеженнями, зокрема

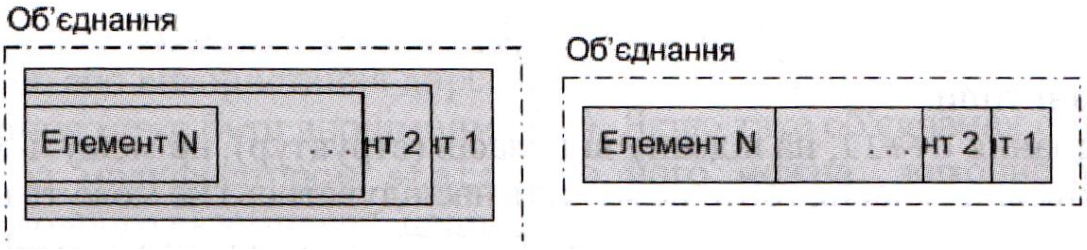
- забороняється отримувати адресу бітового поля та його розмір;
- БП не можуть бути елементами масивів;
- БП не можуть бути статичними.

8.2. Об'єднання

Об'єднання – це структура даних, елементи якої розміщені по одній адресі. Синтаксис оголошення є подібним до структур

```
union [ім'я_об'єднання]
{
тип    ім'я_ елемента1;
тип    ім'я_ елемента2;
...
тип    ім'я_ елементаN;
} [список_змінних];
```

Адреса розміщення екземпляра об'єднання визначається полем, тип якого є найбільшим за розміром. Розміром цього поля також визначається розмір самого



а) логічне б) фізичне

Рис. 8.1. Представлення об'єднання

об'єднання. На Рис. 8.1, подібно до випадку структури, наведено логічна і фізична структура відображення представлення об'єднання його полями. На ній елемент 1 є найбільшого за розміром типу. А усі решта елементів вирівняні по його границі. Наприклад

```
union U
```

```
{
int i;
double d;
} u;
```

Екземпляр об'єднання u є змінною розміром sizeof(double). За допомогою його інтерфейсів (полів) i і d можна проводити операції читання/запису даних у форматах:

- чисел з плаваючою крапкою задіюючи усю ділянку (інтерфейс d).
- додатних чисел типу int, задіяючи лише підділянку довжиною 4 байти (інтерфейс i).

З точки зору користувача об'єднання u зберігає або ціле число i або дійсне число d. Цим самим можна забезпечити операцію конвертації типів. Наприклад за допомогою екземпляра наведеного об'єднання U можна записати ціле число засобом інтерфейсу i, а зчитати його можна за допомогою інтерфейсу d дійсним числом. У результаті отримано швидку реалізацію операції перетворення типів, але за повною відсутності будь-яких засобів контролю типів.

За допомогою об’єднання можна організовувати одночасно доступний різноманітний доступ до однієї і тієї ж ділянки пам'яті. Тобто, об'єднання дають змогу будувати дуже гнучкі інтерфейси, які, у додаток, будуть володіти великою швидкодією. Типовим прикладом є побудова користувацького типу для представлення значення 4-х байтового кольору, наприклад палітри ARGB

```
union ARGB
{
int i;
char m[7];
} argb;
```

Використовуючи це об'єднання, можна маніпулювати значенням 4-х байтового цілого у форматі цілого числа (інтерфейс i), або за складовими кольору - значеннями червоного, зеленого, синього кольорів та альфа каналу (інтерфейс m).

Оскільки об'єднання є різновидом структури мови C++, то більшість тез, мають місце у випадку об'єднання. Зокрема організація доступу до полів об'єднань, можливість вкладень, наявність методів і інші. Відмінними є окремі властивості, які зумовлені способом розміщення змінної в пам'яті. Тому зупинимось на них детальніше.

Зважаючи на те, що усі дані, які є членами об'єднання знаходяться в одній області пам'яті, стало можливим створювати такі типи класів, у яких усі дані знаходяться в одній області пам'яті.

Типом поля у об'єднаннях C++11 можуть використовуватись не лише POD типи, а й агреговані типи.

Об'єднання C++11, на відміну від класів (структур) не можуть мати:

- батьківських класів, тобто не можуть наслідуватись і не можуть наслідувати.
- статичних членів та об'єктів з конструкторами та деструкторами. Якщо член об'єднання містить користувацький конструктор, деструктор чи оператори копіювання то ці функції будуть видалені;
- посилань.

Механізми вкладень та вирівнювання адрес також мають місце у випадку об'єднання. Наприклад, розглянемо код із заданим режимом вирівнювання по довжині машинного слова (4 байти)

Приклад № 8.1.

```
#include <iostream>
using namespace std;
union U
{
struct A
{
int i; // поле вкладеної структури
char c; // поле вкладеної структури
int j; // поле вкладеної структури
} a; // поле об’єднання
char m[7]; // поле об’єднання
} u; // екземпляр об’єднання
void main(void)
{
cout << "Size of U: " << sizeof (U) << endl;
cout << "Size of U::A :" << sizeof(U::A) << endl;
cout << "Size of u.m: " << sizeof(u.m) << endl;
cout << "Size of u.a: " << sizeof(u.a) << endl;
}
```

У тілі об'єднанні U оголошено вкладе структуру, яка, з одного боку, використовується д визначення поля a цього об'єднання, а з іншого як окремий тип, ідентифікатор якого доступний з явним використанням оператора ОДД.

Фізичне розміщення в пам'яті змінної u типу U наведене на Рис. 8.2. Зауважимо, що на цьому рисунку змінні i,c, j та m розміщені в одній ділянці пам'яті.

З наведеної схеми видно, що за рахунок вирівнювання в екземпляр структури було додано три сторінкових байти (paging bytes). У результаті цього поле m перекриває байти, які явно не належать полю a. Це потенційно може

призвести до помилок трактування структури. Виходом із цієї ситуації є або врахування вирівнювання, або використання директиви #pragma pack.

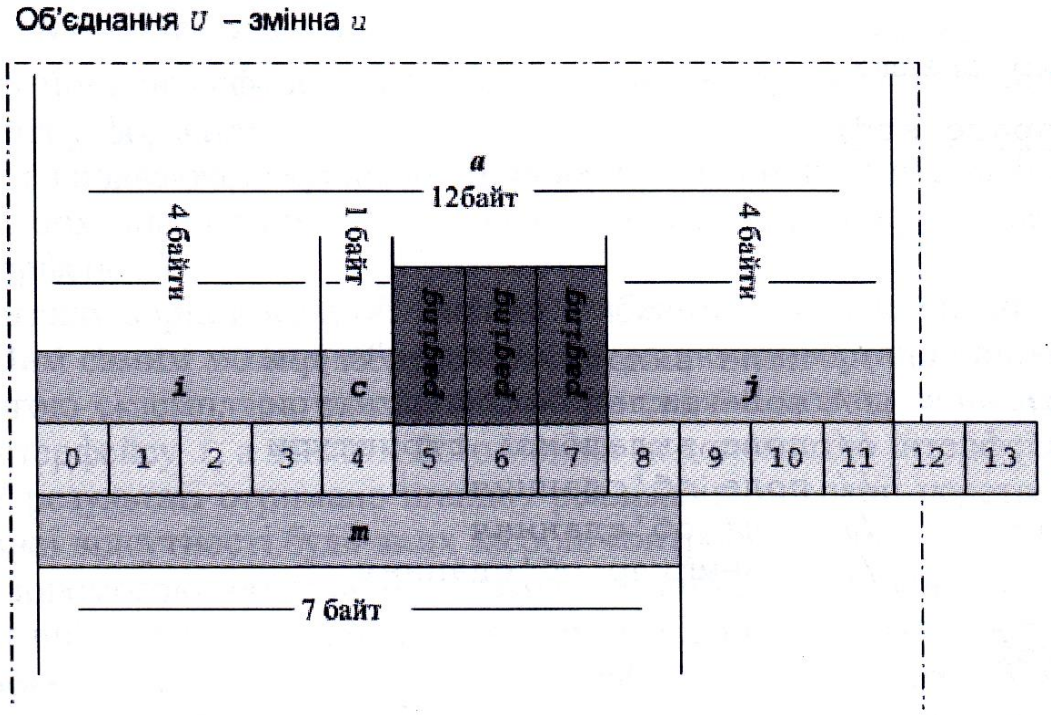


Рис. 8.2. Фізичне розміщення у пам'яті полів змінної u з Прикл. № 8.1

Об’єднання може бути анонімним (АО). Якщо таке об'єднання оголошується глобальним із модифікатором static, то його можна використовувати без створення жодного екземпляра, наприклад

```
static union        // оголошення глобального АО
{
        int x;
} ;                 // екземпляр не створюється
...
x++;                // доступ до поля
::x++;              // доступ до поля з анонімним ОДД
```

Якщо анонімне об'єднання оголошується в межах локальної області дії, то незалежно від результату створення екземпляра об'єднання використання модифікатора static є необов’язковим.

Приклад № 8.2.

Файл - var.cpp

```
#include "var.h"
int y; // змінна класу пам'яті extern
static union // оголошення глобального АО
```

```
{
int x;
// int get_su() {return 1;} - метод створювати заборонено
} ; // екземпляр не створюється
union // оголошення глобального АО
{
int x;
int z;
int get_u() {return x;} // - метод створювати дозволено
} u; // екземпляр створюється
// інтерфейс доступу до полів глобальних АО із інших файлів
// лише для зчитування
int Get_su(){return at_su();}
int Get_u(){return u.get_u();}
// повний доступ
int& at_su(){return x;} // доступ до поля АО
// можливий доступ з анонімним ОДД, наприклад
//int& at_su(){return ::x;}// доступ до поля АО
int& at_u(){return u.x;} // доступ до поля АО
//int& at_uz(){return z;} // помилка доступу до поля
```

Файл - var.h

```
#ifndef VARH
int& at_su();
int& at_u();
int Get_su();
int Get_u();
extern int y;
#define VARH
#endif
```

Файл - run.cpp

```
#include <iostream>
#include "var.h"
using namespace std;
void main(void)
{
static union // оголошення локального АО
{
int r;
// int get_u() {return r;} - метод створювати заборонено
} ; // екземпляр не створюється
```

```
union // оголошення локального АО
{
int r;
int get_lu() {return r;} // - метод створювати дозволено
} lu;// екземпляр створюється
// ініціалізація полів локальних АО
r = -1; // доступ до поля АО
lu.r = -2; // доступ до поля АО
//x = 1; // доступу немає
// ініціалізація полів локальних АО
// які оголошувались у інших фалах проекту
at_su() = 1;
at_u() = 2;
cout << "Local Anoniumos Union field: " << r << endl;
cout << "Local Anoniumos Union (created variab.) field: "
<< lu.get_lu() << endl;
cout << "Global var - : " << y << endl;
cout << "Anoniumos Static Union field: " << Get_su() << endl;
cout << "Anoniumos Non Static Union (created variab.) field: "
<< Get_u() << endl;
}
```

Приведені у прикладі коментарі дають роз'яснення коду. Додамо лише таке. Модифікатор static при визначенні глобального анонімного простору імен без створення екземпляру визначає ля цього об’єднання як глобальні інні класу пам'яті static. Тому я організації доступу до таких інших із інших файлів створюють допоміжні інтерфейси (нашому випадку це функції at_su(), at_u(), Get_su(), Get_u()). Відповідно помилкою буде створення в межах файлу глобального анонімного об'єднання з полями, імена яких збігаються з іменами глобальних змінних, наприклад

```
#include <iostream>
using namespace std;
static union // оголошення глобального АО
{
int x;
} ;
void f()
{
::x = 2; // доступ до x з об'єднання
}
namespace // оголошення АПІ
```

```
{
int x;
void f() { x = 3;}
int outU() { return ::x; } // повертає x з об'єднання
int outN() { return x; } // повертає x з простору
int& at(){ return x; } // доступ до x з простору
}
void main(void)
{
::f();
cout << "Namespace: " << outN() <<
" Anoniomous union: " << outU() << endl;
// x = 4; доступ заборонений
        ::x = 5; // доступ до x з об'єднання
at() = 4; // доступ до x з простору
cout << "Namespace: " << outN() <<
" Anoniomous union: " << outU() << endl;
}
```

У наведеному коді звертання у форматі x = 4; спричинить помилку періоду компіляції, оскільки не зрозуміло якої саме змінної (з об'єднання чи простору) це присвоєння стосується. Оскільки для доступу до поля анонімного об'єднання можна використати анонімний ОДД, то для організації доступу до змінної x з анонімного простору імен довелось створити додаткову функцію at().

Подібно до структур в об’єднаннях також можуть БП. Їх основне призначення створення так званих об'єднань з прапорцями, наприклад

Приклад № 8.3.

```
#include <iostream>
union U
{
int t:1; // оголошення БП як прапорця
struct A
{
int t:1;
int i;
} a;
struct B
{
int t:1;
int i;
} b;
```

```
} u;
bool Is(U u)
{
return (u.t)? false : true;
}
void main(void)
{
u.t = 1;
( Is(u) ) ? u.a.i = 10 : u.b.i = 20;
std::cout << u.b.i;
}
```

Об'єднання може бути частиною структури. Для створення екземпляра такого об'єднання треба пам'ятати, що його ідентифікатор має областю дії тіло структури. Наприклад

```
struct S
{
   union U
{
   int a;
};
   int b;
};
S::U u;
```

З іншого боку поле структури може бути екземпляром вкладеного об'єднання, наприклад

```
struct S
{
   union U
{
   int a;
} u; int b;
};
```

Тоді створення екземпляра структури автоматично призведе до створення копії об'єднання, наприклад

```
S* p = new S;
p->u.a = 1;
```

Об'єднання у структурі також може бути анонімним. При цьому створювати екземпляр такого об'єднання в самій структурі є необов'язково:

```
struct S
{
union
{
int a;
};
int b;
} ;
```

Доступ до полів об'єднання є прямим із екземпляра об'єднання

```
p->a = 1;
```

Зворотній випадок, тобто розміщення структури чи іншого користувацького типу в об'єднанні, у мові C++ накладав на обмеження на типи з нетривіальними конструкторами. У версії C++11 цього обмеження немає, а тому допустим є такий код:

```
typedef struct Point
{
char x, y;

Point() {}
Point(char X, char Y): x(X), y(Y) {}
} PNT;

union U
{
short v;
////// Допустимий код лише у C++11          ////
PNT p;          // Тип з нетривіальним конструктором
U() { new( &p ) PNT(); }
};
```

Для перевірки типу об'єднання бібліотека <type_traits> має шаблон template< class T > struct is_union, наприклад

```
#include <iostream>
```

```
#include <type_traits>
struct A {}; typedef union
{
int a;
float b;
} B;
struct C
{
B d;
};
void main(void)
{
std::cout << std::boolalpha;
std::cout << std::is_union<A>::value << '\n';
std::cout << std::is_union<B>::value << '\n';
std::cout << std::is_union<C>::value << '\n';
std::cout << std::is_union<int>::value << '\n';
}
```

Лекція 9. Вказівники.

Вказівники відігравали і відграють фундаментальну роль в мовах C та C++). Їхнім призначенням є пряме забезпечення адресних операцій. Якщо уважно подивитись на код будь-якого прикладного програмного рішення, написаного на мові C/C++, то можна переконатись, що вказівники у ньому використовуються всюди. А тому без розуміння вказівників неможливо зрозуміти мови C та C++.

Написання програми є суцільним маніпулюванням адресами в області пам'яті, тобто маніпулювання вказівниками. Більшість цих операції здійснюються в контексті змінної і саме цим приховуються адресні операції. Відповідно використання змінних вважається прямим звертанням до пам'яті.

Напротивагу цьому існує непряме звертання, яке реалізовується, зокрема у мові C++, засобами двох конструкцій: вказівників та посилань. За своєю суттю вони є носіями адреси інформації, яка зберігається в пам'яті.

9.1. Вказівники

Вказівник (pointer) – спеціальна змінна, в якій зберігається адреса іменованих (змінні) чи неіменованих комірок пам’яті.

З формальної точки зору вказівники дозволяють

- обробляти динамічні структури даних;
- маніпулювати пам’яттю, стрінгами та масивами;
- ефективно передавати аргументи у функцію через параметри самої функції. Володіння адресою дає вказівникові можливість отримати доступ до значення, яке зберігається за даною адресою. Саме тому звертання до даних за допомогою вказівників (та посилань) називається непрямим.

Мова C++ дає можливість оголошувати вказівники двояко:

```
char *name; char* name;
```

Ці оголошення є повністю рівносильними. В обох випадках створено вказівник на змінну типу char, яка зберігається у деякому місці в пам'яті. Як правило, перше оголошення використовується для одночасного створення декількох вказівників, а друге використовується в прототипах функцій, де оголошуються лише типи без змінних

```
void f_A ( char*, int, short* );
```

Загальний синтаксис для оголошення вказівника і ініціалізація його адресою існуючою змінною має вид:

```
type variable;
[modifiers] type [modifiers] * [modifiers] pointer [ = &variable];
```

У приведеному синтаксисі оголошення модифікатори та ініціалізація вказівника є необов'язковими.

При оголошенні декількох вказівників знак '*' ставиться біля кожного ідентифікатора.

Нульове значення (нульова адреса) для вказівникового типу задається константою NULL. Саме значення NULL приймається хибним у логічних операціях.

Вказівник проініціалізований значенням NULL вважається нульовим або порожнім.

Оскільки реалізовується константа NULL макропідстановкою, то у загальному випадку у різних реалізаціях C++ значення константи може бути різним. Тобто існує виникає проблема літерального значення, яке буде закладатись у цю константу.

Однозначного вирішення проблеми оптимального значення для визначення константи NULL не не існує. Типово у мові C літеральне значення NULL є рівним ((void*)0) , а у мові C++ є рівним 0. Це породжує проблеми використання константи NULL, оскільки остання може розглядатись як ціле число так і нульовий вказівник. Найбільш яскравим прикладом проблеми константи NULL є такий код

```
void foo(char *);
void foo(int);
```

Відповідно виклик foo(NULL); у мові C++ приведе до виклику функції foo(int), а не foo(char *).

Для вирішення проблеми нульового вказівника у C++11 пропонується використовувати константу nullptr, яка є типу std::nullptr_t. Цей тип неявно приводиться до типу будь-якого вказівника і може використовуватись у визначених для них операторах порівняння. За винятком типу bool забороненим є неявне приведення до цілого типу.

```
char *p1 = nullptr;
int     *p2 = nullptr;
bool   b = nullptr;   // правильно b = false.
int     i = nullptr;    // помилка - приведення заборонене
```

Виклик foo(nullptr); гарантоване забезпечить виклик foo(char *) , а не foo(int).

Константи nullptr і NULL можуть використовуватись сумісно.

Тип, на який посилається вказівник може бути довільним преозначеним типом, зокрема може бути і вказівником. Тут важливим є те, що тип, який використовується для оголошення вказівників повинен бути наперед визначеним

(преозначеним). Це дає можливість компілятору здійснювати перевірку типів, занесення інформації по адресі, проведення операцій над вказівниками та значеннями, на які вони посилаються.

Поняття типу вказівника не існує, а існує лише поняття типу даних, на які посилається вказівник. У процесі типізації тип вказівника визначається як тип на, який посилається вказівник. Іншими словами – після оголошення змінної int *j, її типом вважається вказівник на int, тобто int* . Це означає, що вказівники на тип int і double є формально змінними різних типів, незважаючи на те, що обидва є вказівниками і під виділяється однаковий розмір пам'яті.

Вказівник на преозначений тип вважається типованим вказівником. Поряд з ним існує так званий безтиповий вказівник, або void-вказівник, оголошення якого є таким: void *j.

Тип в означенні вказівника відіграє фундаментальну роль. У першу чергу, він є необхідним для організації посилань на комірки пам'яті в процесі роботи оператора розіменування.

У випадку типованого вказівника розміром типу, на який він посилається, визначається мінімальний обсяг пам'яті (МОП), який адресується цим вказівником. Цей мінімальний розмір задається у байтах і визначає так звану умовну одиницю (УО) для типу вказівника. Тому типований вказівник завжди повинен розглядатись у контексті пари значень: адреса: МОП. Для безтипового вказівника МОП не визначено. відповідно не визначена умовна одиниця для цього типу. Це означає, що безтиповий вказівник повинен розглядатись виключно в контексті адреси.

Якщо типовані вказівники разом із операцією доступу є визначеними, стосовно типу, інтерфейсом доступу до пам'яті, то безтипові вказівники забезпечують доступ до даних будь-якого типу.

Подібно до звичайної змінної перед першим використанням вказівник потребує ініціалізації. Ініціалізація вказівника – це присвоєння йому адреси існуючої комірки пам'яті або щойно виділеної. У випадку глобального вказівника ініціалізатором по замовчуванні виступає присвоєння нульового значення.

У випадку локального вказівника ініціалізації немає, а тому вказівник містить довільне випадкове значення, яке може бути наслідком попередньої операції.

В обидвох випадках існує можливість визначити ініціалізацію вказівника явно при його оголошенні

```
int i = 3;
int *j  = &i;
```

Змінити значення вказівника можна в процесі виконання програми. Для цього визначеною є операція присвоєння, наприклад

```
int *j;
```

```
j = &i;
```

У випадку використання вказівників розглядаються два види операторів присвоєння. Перший з них стосується самих вказівників. У цьому випадку формально оператор присвоєння є визначеним для вказівників однакового типу. Тобто валідною є операція

```
int *p1, *p2;
...
p1 = p2;       // допустима операція
```

і не валідною є операція присвоєння для вказівників різного типу

```
double *p3;
p1 = p3;       // недопустима операція
p1 = (int*)p3;        // допустима операція
```

Послідовність дій у випадку валідної операції присвоєння є такою. На початку l-вираз, який визначається ідентифікатором вказівника, повертає посилання на пам'ять, яка під нього була виділена. Це посилання виступає аргументом оператора присвоєння. Якщо цей аргумент є лівим операндом (lvalue, у наведеному прикладі це p1), то він буде повертатись як результат виконання оператора присвоєння. Вміст комірки, яка адресується цим посиланням, буде мати модифіковане оператором присвоєння значення. У нашому випадку вказівник p1 буде мати таке ж значення (у трактуванні адреси), яке мав вказівник p2.

Для забезпечення операції присвоєння між вказівниками різного типу потрібно попередньо виконати операцію присвоєння у явній формі. У цьому випадку оператором приведення буде створений новий тимчасовий вказівник із таким значенням, яке містив вказівник, що приводився. Оскільки отриманий вказівник є тимчасовою змінною, то посилання на нього не можна повертати. Це означає, що він може виступати лише в якості rvalue оператора присвоєння.

Присвоювати літеральні значення вказівникам заборонено. Проте подібно до випадку вказівників різного типу, можна використати операцію приведення типу і вирішити це завдання, наприклад

```
double *p = 0xa1;                 // недопустима операція
double *p = (double)0xa1;         // допустима операція
```

Відокремлено розглядається присвоєння для випадку void-вказівників. Присвоювати void-вказівнику можна значення будь-якого вказівника: void *p4 = p1;. А от зворотна операція потребує приведення типу у явній формі: p1 = (int*)p4;. Виключення становлять значення, які повертають оператори виділення

пам'яті new та взяття адреси &. Ці оператори повертають void* вказівник. Проте явне приведення типу у цьому випадку є зайвим, оскільки буде задіяним механізм неявного приведення.

Другий вид оператора присвоєння стосується комірок пам'яті, які адресуються вказівниками. У цьому випадку оператор присвоєння визначається типом комірок, які адресуються вказівниками. Для отримання посилання на ці комірки використовується оператор розіменування. Він реалізовує операцію доступу до значення, яке знаходиться в пам'яті за адресою, яку містить вказівник, наприклад

```
...
int i = 3;
int *j = &i;          // зміна буде додатково адресуватись вказівником
std::cout << i;       // i = 3
*j = 2;               // використання вказівника
std::cout << i;       // i = 2
```

Оператор розіменування, подібно до випадку звичайної змінної, також l-виразом. Значенням lvalue, яке він повертає, є посилання на комірку пам'яті, яке адресується цим вказівником. Наприклад запис *j; поверне посилання на комірку пам'яті, яка виділялась під змінну i, у форматі int&.

Якщо тип, на який оголошується вказівник, є структурованим, то операція розіменування використовується сумісно з оператором доступу, наприклад

```
struct A {int i;} a;
A *p = &a;
*p.i = 1; // оператори розіменування та доступу
```

Для структурованих типів допускається використання єдиного (разом із розіменуванням) оператора доступу 'стрілка'

```
*p.i = 1; або p->i = 1;
```

Через відсутність типу є неможливим визначення lvalue для void-вказівників. Тому операція розіменування для безтипових вказівників є невизначеною. Проте це обмеження можна обійти шляхом явного приведення безтипового вказівника до типованого, наприклад, якщо взяти за основу визначення вказівника j у попередньому прикладі, то отримати значення через безтиповий вказівник можна так

```
void *p = j;
//int n = (int)(*p); - помилка - недопустима операція
int n = *( (int*)(p) );
```

У результаті сказаного вище розглянуто дві базові операції із вказівниками, а саме: присвоєння та розіменування. Окрім них над вказівниками, але лише типованими, можна здійснювати арифметичні операції додавання та віднімання.

Це забезпечується існуванням УО. Розглянемо це на прикладі

```
double i; double *p = &i;
p - 1;
p++;
p += 2;
```

Умовна одиниця для типу double* є рівна sizeof(double). Тому в усіх випадках отримаємо зміну значення адреси, яку містив вказівник p. Фактично наведений код є еквівалентний такому

```
(int)p - sizeof(double);
p = (double*)( (int)p + sizeof(double) );
p = (double*)( (int)p + 2*sizeof(double) );
```

У арифметичних операціях над вказівниками треба чітко розрізняти l-вирази, оскільки від цього залежить зміна значення самого вказівника. У наведеному коді у випадку інструкції p - 1; значення вказівника p не зміниться.

Для безтипових вказівників умовна одиниця не визначена. Тому для них операції додавання та віднімання є недопустимими.

C++ допускає використання вказівників з модифікатором const. Це дає можливість оголосити:

а) постійним значення, на яке посилається вказівник. Відповідно забороненим буде зміна значення, яке адресується вказівником

```
int k = 5;
const int *p = &k;  або    int const *p = &k;
...
// (*p)++; - помилка зміна адресованого значення є забороненою
k++;
```

б) постійним вказівник. Такий вказівник називається констатним. У випадку цього вказівник забороненою є зміна значення самого вказівника

```
int k = 5;
int * const p = &k;
...
//p = new int; - помилка зміна значення вказівника є забороненою
```

```
(*p)++;
```

в) обидві сутності (адресоване значення і вказівник) постійними об'єктами. Такий вказівник використовується виключно для читання значення

```
int k = 5;
const int * const p = &k;  або    int const * const p = &k;
...
//p = new int; - помилка зміна значення вказівника є забороненою
// (*p)++; - помилка зміна адресованого значення є забороненою
int n = *p;
```

Використання модифікатора const є одним з видів захисту значення вказівника та значення на яке він посилається. Проте як не дивно, це захист дуже легко обійти, оскільки на етапі компіляції контролюються явно визначені операції зміни значення. наприклад

Приклад № 9.1.

```
#include <iostream> using namespace std;
const int i = 11; // глобальні констатнти
const int * const pi = &i;
void print_results ( char* info, const int& j, const int * const pj, const int * const
p_i )
{
cout << info << endl
<< "\t in       **print_results**  function " << endl
<< "\t\ti -     " << i << "\t\t*pi - " << *pi        << endl
<< "\t\t&i - " << (int)&i << "\tpi - " << (int)pi <<endl
<< "\t\tj -     " << j << "\t\t*pj - " << *pj        << endl
<< "\t\t&j - " << (int)&j << "\tpj - " << (int)pj <<endl
<< "\t\t*p_i - " << *p_i   << "\tp_i - " << (int)p_i <<endl;
}
void main(void)
{
const int j = 11; // локальні констатнти
const int * const pj = &j; const int * const p_i = &i;
print_results("Initial values: ",j , pj, p_i ); cout << "\t in     **main**    function "
<< endl
<< "\t\tj -     " << j << "\t\t*pj - " << *pj        << endl
<< "\t\t&j - " << (int)&j << "\tpj - " << (int)pj <<endl;
// *(int*)pi = 2; - помилка періоду виконання
//  *(int*)&i = 2; - помилка періоду виконання
```

```
//  *(int*)&p_i = 2; - помилка періоду виконання
*(int*)&j = 2;
print_results(" Modification #1: ", j , pj, p_i ); cout << "\t in       **main**
function " << endl
<< "\t\tj -     " << j << "\t\t*pj - " << *pj        << endl
<< "\t\t&j - " << (int)&j << "\tpj - " << (int)pj <<endl;
*(int*)pj = 3;
print_results(" Modification #2: ", j , pj, p_i ); cout << "\t in       **main**
function " << endl
<< "\t\tj -     " << j << "\t\t*pj - " << *pj        << endl
```

Інший спосіб деактивації дії модифікатора const полягає у використаня операторів динамічної ідентифікації типів, зокрема const_cast.

Типовим використанням вказівника є організація захищених інтерфейсів доступу до пам'яті. Тобто таких, які забезпечують лише операції читання наприклад

```
...
int i;
...
const int* p = &i;
i++;                    //дозволена операція
(*p)++;                 // заборонена операція
```

Модифікатор volatile, подібно до const, також можна використовувати із вказівниками. При цьому також можливі три випадки: вказівник на volatile-змінну, volatile-вказівник, volatile-вказівник на volatile-змінну, наприклад

```
volatile int i;
int j;
volatile int* p1 = &i;
p1 = &j;
int * volatile p1 = &j;
p2 = &i; // заборонена операція
volatile int * volatile p3 = &i;
p3 = &j;
```

Нарешті мова C++ дозволяє сумі використання модифікаторів volati і const. Тут можливими є багато ком націй сумісного використання. Серед н принциповими є три такі комбінації

```
const volatile int i;
```

```
const int j;
volatile int m;
int n;
const volatile int* p1 = &i;
p1 = &j;
p1 = &n;
p1 = &m;
int * volatile const p2 = &n;
p2 = &i;            // заборонена операція
p2 = &j;            // заборонена операція
p2 = &m;            // заборонена операція
const volatile int * const volatile p3 = &i;
p3 = &j;            // заборонена операція
p3 = &n;            // заборонена операція
p3 = &m;            // заборонена операція
```

Мова C++ дозволяє оголошувати вказівник на вказівник (подвійний вказівник). Загальний синтаксис має вигляд

```
[modifiers] type [modifiers] ** [modifiers] pointer [ = &variable];
```

Наприклад,

```
const int i = 11; const int * pi = &i;
// int * pi = &i; - помилка
const int * const p = &pi;
```

Операція подвійного знаходження адреси &&, яка інтуїтивно напрошується стосовно подвійного вказівника, є недопустимою, тобто наступний код є неправильним

```
int i = 1;
int ** j = &&i;             // недопустима операція
```

Це пояснюється тим, що результатом l-виразу & i є посилання у форматі тимчасової змінної. А за правилами мови операція взяття адреси від тимчасової змінної є забороненою.

Розіменування подвійного вказівника до типованої змінної здійснюється через подвійну операцію розіменування ** . Наприклад

```
a) int i = 5; int * j = &i; int ** z = &j;
**z = 4;        // розіменування подвійного вказівника
```

Зазначимо, що більше двох рівнів непрямої адресації, в переважній більшості випадків не використовується. Проте різні компілятори дають різну кількість рівнів організації багаторівневих ланцюжків непрямої адресації. Так, наприклад Visual C++ 6.0 дозволяє організовувати до 1010 рівнів, а Turbo C++ дозволяла 70 рівнів.

При визначенні вказівників допускається використання операторів decltype та auto. Наведемо приклади використання цих операторів

```
auto y = 10L;              // z - є типу long
const auto *p = &y;        // p є const double *
decltype (p) p1 = p;       // p1 є const double *
```

Для перевірки типу об'єднання бібліотека <type_traits> має шаблон template< class T > struct is_pointer, наприклад

```
#include <type_traits>
#include <iostream> struct A {};
...
std::cout << std::boolalpha
        << std::is_pointer<A>::value << '\n'
        << std::is_pointer<A*>::value << '\n'
        << std::is_pointer<float>::value << '\n'
        << std::is_pointer<int>::value << '\n'
        << std::is_pointer<int*>::value << '\n'
        << std::is_pointer<int**>::value << '\n';
```

9.2. Довгі та короткі вказівники

Логічна адресація на платформах x86 здійснюється у форматі сегмент:зміщення. За допомогою пари цих значень отримується фізична адреса. При цьому одній парі значень сегмента та зміщення буде відповідати одне ціле значення фізичної адреси. Проте одній фізичній адресі можуть відповідати декілька різних пар значень сегмент:зміщення. Ця ситуація називається ал'ясингом пам'яті).

Модель пам'яті (МП) для платформ x86 – це вказівки компіляції при генерації коду для обчислювальних платформ із сегментною (segmented memory model), сторінковою (paged memory model) або лінійною адресацією пам'яті (flat memory model).

Сегментація, яка була основним підходом до організації пам'яті на 16-и розрядних платформах, може використовуватись і на 32-х розрядних платформах. В її основі лежить використання сегментних регістрів: DS (data

segment) – сегмент даних; CS (code segment) – сегмент коду; SS (stack segment) – сегмент стеку; ES (extra segment) – додатковий сегмент та SP (stack pointer) – регістр границі заповнення стеку. Використання цих регістрів забезпечує організацію адресацію за допомогою коротких (або ближніх, near) вказівників. Далека адресація базується виключно на використанні довгих далеких far-великих huge-вказівників.

Поява сторінкової адресації пам'яті знівелювала поняття довгого і короткого вказівника. Усі вказівники стали довгими і виражають 32-х розрядні віртуальні адреси навіть на 64-х розрядній платформі.

У плоскій моделі адресації, яка в одному адресному просторі зберігає дані і код, вказівники також усі довгі. Їх розрядність безпосередньо визначаються розрядністю обчислювального середовища.

На платформі x86-64, яка охоплює 16-и, 32-х та 64-х розрядні апаратно-програмні середовища, в залежності від способу адресації існує сім моделей пам'яті. У кожній із цих моделей існує свій вид (і, відповідно розмір) вказівника по замовчуванні. Їх перелік також наведені у таблиці додатку.

Короткий (ближній, тип near) вказівник зберігає лише зміщення відносно початку даного сегменту. Тому він може адресувати комірки пам'яті у сегментах DS та CS. Відповідно регістри, які адресують ці сегменти, не повинні змінювати своїх значень в процесі розіменування вказівника.

Довгий далекий вказівник (тип far) містять адресу у повному форматі сегмент-зміщення. Це визначає більші витрати пам'яті під зберігання значення такого вказівника. Для використання far-вказівників значення регістра змінюється, здійснюється розіменування і, накінець, відновлюється значення регістра.

Великі вказівники (huge) є подібними до far-вказівників, але вони нормалізуються після кожної їх зміни. У плоскій моделі пам'яті (модель flat) це дає змогу уникнути ал'ясингу пам'яті і, відповідно, точнішою є операція порівняння вказівників.

Стандарт C++11 не підтримує зарезервованих слів near, far та huge оскільки вид вказівника визначається компілятором автоматично в залежності від платформи призначення програмного продукту, який розробляється.

.

Лекція 10. Керування пам’яттю.

10.1. Вказівники і динамічні змінні (керування пам'яттю)

Будь-яка фізична сутність (змінна, об'єкт) може статичною, автоматичною або динамічною. Перша з них визначається на етапі компіляції і розміщується пам'яті під час запуску програми (породження процесу).

Автоматична сутність розміщується в пам'яті кожного разу, коли управління входить в цей блок, і знищується, коли управління виходить з цього блоку.

Динамічні об'єкти є такими, які створюються явно користувачем у процесі виконання коду. Тривалість їх існування також визначається користувачем у явній формі. Такі об'єкти створюються за допомогою операції new і знищуються за допомогою операції delete. Підтримується використання бібліотеки malloc.h, у яких визначено бібліотечні функції виділення та звільнення пам'яті.

Розміщуються об'єкти у вільній у адресному просторі пам'яті. Через існування певної організації роботи з цією пам'яттю, остання отримала спеціалізовану назву – купа (heap) . У будь-який момент періоду існування купи, динамічна пам'ять ділиться на зайняту і вільну. Зайнятою вважається попередньо виділена і до цих пір ще не звільнена пам'ять. Вільною вважається пам'ять, яка у відповідь на запит виділення певного обсягу пам'яті, за допомогою примітивів для роботи з купою може бути переведеною в категорію зайнятої. Відповідно суть процесів виділення та звільнення пам'яті полягає у переведенні пам'яті купи із однієї категорії в іншу.

Фізично купа – це довгий відрізок адрес пам'яті, який розділений на послідовні блоки різних розмірів. Ці блоки можуть бути помічені або вільні, або зайняті.

Для відмітки про належність блоку пам'яті до зайнятої чи вільної використовується додаткова область пам'яті, яку називають картою пам'яті (купи). Карта пам'яті може містити одну чи декілька таблиць (списків) адрес вільних блоків.

Карта купи може бути організована по різному. Від її організації безпосередньо залежить продуктивність купи, оскільки суть процесу виділення пам'яті полягає в аналізі списку вільних блоків в карті пам'яті зокрема пошуку вільних блоків у таблиці пам'яті.

Для зменшення розміру списку вільних блоків послідовні вільні блоки, іноді, об'єднують в один. Якщо вільним є наступний блок, то його віднайти можна шляхом пересування на розмір блоку, який звільняється. Для пошуку попереднього блоку, інформація про нього (розмір), зберігається у заголовку кожного поточного блоку.

З метою збільшення швидкодії карта купи може містити декілька списків, кожен з яких орієнтований на певний розмір блоку. Це дозволяє ігнорувати цілі набори блоків малої довжини без індивідуального аналізу кожного.

Перед початком виконання коду програми здійснюється ініціалізація купи. У процесі цієї ініціалізації уся пам'ять, яка виділялась під купу, помічається як вільна.

Змінні, які розміщуються в купі, називаються динамічними. У переважній більшості вони є неіменованими. Доступ до них забезпечується вказівниками чи посиланнями (постійними вказівниками).

Вказівники, завдяки своїй гнучкості є основним інструментальним засобом для роботи з купою і, відповідно, динамічними змінними. Вони реалізовують багатоцільовий механізм. З одного боку використовуються для непрямої адресації існуючих змінних, які розміщені в області стеку (сегмент SS) чи в області даних (сегмент DS), і забезпечують цим альтернативний доступ до їх значень. З іншого – вказівники дозволять керувати розподілом області динамічної пам'яті. У загальному випадку, якщо брати до уваги сегментний спосіб адресації, може існувати два види куп:

- локальна (або ближня) купа – local heap. Ця купа формується із вільної пам'яті в сегменті DS,
- глобальна (або далека) купа – global heap. Ця купа формується з вільної області адресного простору, який виділяється процесу, що породжений програмою.

У різних моделях пам'яті програм можуть бути різні співвідношення існування локальної та глобальної куп. Зокрема від повної відсутності куп (модель tiny) до існування лише глобальної (моделі flat чи huge). Адресація в локальній купі здійснюється лише за допомогою ближнього вказівника, оскільки адреса сегмента, в якому розміщена купа, зберігається в регістрі DS. Використання далекого вказівника можливе (помилку може виправити компілятор мови C++, але не C), але не виправдане.

Адресація в глобальній купі може здійснюватися як через ближні так і через далекі вказівники (в окремих випадках компілятор мови C++ самостійно може перевести вказівник з класу ближніх в клас далеких).

Мова C++ орієнтована на використання динамічних змінних виключно в області далекої купи. Керуючи динамічними змінними необхідно:

а) явно створювати і знищувати динамічні змінні та об'єкти;

б) для кожного динамічного об'єкта організовувати принаймні один вказівник, який буде забезпечувати доступу до самого об'єкта.

Як вже відзначалось також підтримується використання бібліотеки malloc.h і, відповідно, бібліотечних функцій ANSI C для роботи з динамічною пам'яттю:

```
// memory allocation, виділення пам'яті
void *malloc (size_t size);
// clear allocation, чисте виділення пам'яті
void *calloc (size_t num, size_t size);
// reallocation, перерозподіл пам'яті
```

```
void *realloc(void *block, size_t size);
// free, звільнення пам'яті
void free(void *block);
```

Стандарт C++ визначає оператори (частину мови) new, new[], delete, delete[] (бібілотека new.h) для виділення динамічної пам'яті. Для цього у синтакси мови введено new-вирази.

(а)::(optional) new type[array_n](optional)(init_params) (optional)
(б)::(optional) new (type) [array_n](optional)(init_params)(optional)
(с)::(optional) new (placement_params) type [array_n](optional) (init_params)(optional)
(в)::(optional) new (placement_params) (type) [array_n](optional)(init_ params)(optional)

Зноска (optional) визначає необов'язковість заповнення позиції при використанні new-виразів. Ці вирази забезпечують виділення пам'яті для масиву чи окремого об'єкта і, у випадку успішного виділення пам'яті, повертають вказівник на перший об'єкт. Останні два вирази, на відміну від перших двох попередніх, називаються виразами з розміщення по адресі (вже виділеної пам'яті), яка задається параметром placement_params. Параметр array_n визначає виділення пам'яті для масиву. Якщо він опущений, то пам'ять виділяється під одиничний об'єкт. Розмір пам'яті визначається розміром типу, які задається параметром type. Допускається початкова ініціалізація значенням за допомогою параметра init_params.

Опційний ОДД напочатку new-виразу визначає пошук функції виділення пам'яті у глобальній області дії. Якщо оператор опущений, то на початку компілятор буде шукати функцію виділення пам'яті у локальній області, наприклад у класі. Лише у випадку не знаходження функції у локальній області, пошук буде продовжено у ГОД.

Виконання new-виразів забезпечується викликом відповідних функцій, повний перелік, яких є таким

```
(1)void* operator new     ( std::size_t count );
(2)void* operator new[]( std::size_t count );
(3)void* operator new ( std::size_t count, const std::nothrow_t&);
(4)void* operator new[]( std::size_t count, const std::nothrow_t&);
(5)void* operator new  ( std::size_t, void* ptr );
(6)void* operator new[]( std::size_t, void* ptr );
```

Тут параметр count задає розмір пам'яті в байтах. Для кожної з цих функцій визначено оператор звільнення пам'яті. Для зручності пошуку відповідного

оператора звільнення оператори виділення пам'яті пронумеровані цифрами у дужках. Відповідний оператор звільнення пам'яті матиме цей ж номер,

```
(1)void operator delete   ( void* ptr );
(2)void operator delete[]( void* ptr );
(3)void operator delete ( void* ptr, const std::nothrow_t& );
(4)void operator delete[]( void* ptr, const std::nothrow_t& );
(5)void operator delete  ( void* ptr, void*);
(6)void operator delete[]( void* ptr, void* );
```

Наведемо приклад використання операторів виділення та звільнення пам'яті

Приклад № 10.1.

```
#include <iostream>
using namespace std;
struct STRC
{
char cmember;
int imember;
STRC(){ cout << "Created " << endl; }
~STRC(){ cout << "Killed " << endl; }
        };
void main(void)
{
// The form #1 of new & delete
try
{
STRC* p1 = new STRC;
cout << "#1 form. Construction STRC - " << p1 << endl;
delete p1;
}
catch(std::bad_alloc)
{
cout << "Memory opertion is failed " << endl;
}
// The form #3 of new delete
STRC* p2 = new( nothrow ) STRC;
if (p2)
{
cout << "#3 form. Construction STRC - " << p2 << endl;
```

```
delete p2;
}
// The form #5 of new & delete
char x[sizeof( STRC )];
STRC* p3 = new( &x[0] ) STRC;
cout << "#5 form. Construction STRC - " << p3 << endl;
cout << "The address of x[0] is : " << ( void* )&x[0] << endl;
p3->~STRC();
// The form #2 of new[] & delete[]
try
{
STRC* p4 = new STRC[2];
cout << "#2 form. Construction massive - " << p4 << endl;
delete[ ] p4;
}
catch(std::bad_alloc)
{
cout << "Memory opertion is failed " << endl;
}
// The form #4 of new[] & delete[]
STRC* p6 = new( nothrow ) STRC[2];
if (p3)
{
cout << "#4 form. Construction massive - " << p6 << endl;
delete[ ] p6;
}
// The form #6 of new[] & delete []
char xm[ 2 * sizeof( STRC ) ];
//char xm[sizeof( STRC )];
STRC* p5 = new( &xm[0] ) STRC[2];
        cout << "#6 form. Construction massive - " << p5 << endl;
cout << "The address of xm[0] is : " << ( void* )&xm[0] << endl;
p5[1].~STRC();
p5[0].~STRC();
        }
```

У випадку підтримки сегментної адресації і, відповідно ближньої і далекої купи, оператор new виділяє пам'ять з глобальної купи, використовуючи функцію GlobalAlloc(). Напротивагу, у цьому ж випадку, функція malloc() використовує функцію LocalAlloc() і виділяє пам'ять в локальній купі. Відповідно оператор new має доступ до більшої по об'єму кількості пам'яті.

У випадку запиту нуля байтів оператор new і функція malloc() (а також функція calloc()) повертають неініціалізований вказівник, наприклад

Приклад № 10.2.

```
#include <iostream>
using namespace std;
void info(int* p)
{
cout << ( (p) ? "Initialized pointer" :
"Not initialized pointer" ) << hex << " - 0x" <<
(int)p << endl;
}
void main(void)
{
int * i = (int*) 1;
info(i);
i = new int[0]; // will return a non-zero pointer
info(i);
i = (int*) 1;
info(i);
i = (int * ) malloc(0); // will return a non-zero pointer
info(i);
        }
```

Це, у свою чергу, може призвести до колізії, оскільки реально для даного програмного об'єкта пам'яті не було виділено.

Використовуючи оператор new можна провести відразу ініціалізацію заданим значенням

```
int * i = new int (0);          //ініціалізація змінної значенням 0
int * i = new int [10] (17);// ініціалізація масиву значенням 17
```

Звільняється пам'ять в купі операторами delete або delete[] для масивів). Зауважимо, що використання оператора delete[] для знищення динамічних масивів за явним вказуванням розміру в дужках було необхідним для ранніх версій C++-компіляторів. Сучасні компілятори позбавлені цього недоліку.

Вказані оператори звільняють пам'ять за адресою, яку містить вказівник, що виступає аргументом. Значення самого вказівника після виконання оператора не міняється. Успішне звільнення пам'яті залежить від значення вказівника. Це означає, що вказівник повинен містити валідну адресу в області динамічної пам'яті.

В противному випадку операція звільнення пам'яті не відбудеться. Формально це відображається відсутністю будь-який подій в період виконання коду.

Валідність значення означає лише значення адресу в області купи і не стосується того чи ділянка пам'яті за цієї адресою є зайнятою. Тому повторні виклики оператора звільнення пам'яті, або знищення за нульовим вказівником є безпечною операцією і не несуть жодної загрози генерування виключень.

Відповідно до сказаного повністю безпечним є такий код

```
int * p = NULL;
...
delete p;      delete (int*)NULL;
p = (int*)1; delete p;
p = new int;
delete p;      delete p; p++;
delete p;      delete[]p;
p = new int [10];
delete[]p;     delete[]p;
p = new int [10];
p++;
delete p;
```

У випадку складних динамічних багатомірних структур, коли елементами масиву є конструкції, які у свою чергу мають власні конструктори та деструктори, лише оператор delete[] правильно звільняє пам'ять. У цьому випадку є необхідним виклик деструкторів для кожного елементу масиву, а оператор delete викликає деструктор лише для одного елемента (того, на який він вказує). Тут треба пам'ятати, що виклик деструктора є окремою дією і не має жодного відношення до процедури звільнення пам'яті. Наприклад

Приклад № 10.3.

```
#include <iostream>
using namespace std;
typedef unsigned int UI;
struct MyStruct
{
char cmember;      int j;
char i;int imember; MyStruct() { cout << "Created; " ; }
~MyStruct() { cout << "Killed " << imember << "; "; }
};
void out(char n, const char* mes, MyStruct* p, UI sz)
{
```

```
cout << endl << "Massive: " << endl; for (UI i = 0; i < sz; ++i)
cout << " i = " << i << " member = " << p[i].imember << endl; cout <<"#" << (int)n
<< mes << " -----------" << endl;
}
int main()
{
MyStruct* p = new MyStruct[5];
for ( UI i = 0; i < 5; p[i].imember = i, ++i );
out( 1, " Before delete [] ", p, 5 ); delete [] p;
out( 1, " After delete [] ", p, 5 );
p = new MyStruct[5];
for ( int i = 0; i < 5; p[i].imember = i, ++i );
out( 2, " New massive is created ", p, 5 );
p += 3;
out( 3, " Moved (incr. 3) pointer. Before delete ", p, 2 );
delete p; delete p;
out( 3, " After delete ", p,2);
// delete[] p;// error of the run-time period
p--;
out( 4, " Moved pointer (decr. 1). Before delete ", p, 3 ); delete p;
out( 4, " After delete and Before delete ", p, 3 ); delete    p;
out( 5, " After delete ", p, 3 );
p -= 2;
out( 6, " Moved pointer (decr. 2). Before delete[] ",p, 5 ); delete [] p;
out( 6, " After delete and Before delete[] ", p, 5 );
// delete[] p;// error of the run-time period
}
```

Їх аналіз пропонуємо зробити самостійно. Особливу увагу треба звернути на помилки викликів оператора delete[].

Важливою перевагою операторів new, delete, delete[] над функціями C є те, що C++ дозволяє визначати нові версії операторів new, delete і delete[], наприклад

Приклад № 10.4.

```
#include <iostream>
using namespace std;
void* operator new( size_t t ) throw()
{
cout << "new operator - parameter - " << t << endl; return NULL;
}
void operator delete(void * p)
{
```

```
cout << "delete operator - parameter " << (int)p << endl;
}
void main(void)
{
int *k;
k = new int; delete k;
k = new int[4]; delete[ ]
k;
        }
```

Наведений приклад ілюструє ще одну можливість C++, яка називається перевантаженням функцій та операторів.

Операції виділення динамічної м'яті можуть призводити до ситуації, яка називається кровотечею пам'яті (memory bleeding) і визначаються наявністю недоступних за формалізованими інтерфейсами доступу динамічних об'єктів. Ці ситуації породжуються процедурами переприсвоєнням вказівникам адрес нових динамічних змінних, при тому, що не ули знищені попередні. Тому треба ого відслідковувати усі динамічні змінні, а у випадку їх невикористання одразу знищувати.

Поведінка оператора new у вихід використовується для усіх дочірніх об'єктів, які входять до складу полів даного типу.

У випадку використання формату new A() стандарт C++11 висуває такі умов:

• якщо тип *A* є POD-класом, то об'єкт буде ініціалізуватись нульовими значенням по замовчуванні;

• якщо тип *A* не є POD-класом, то для нього буде викликатись конструктор по замовчуванні, незалежно від того явно чи неявно цей конструктор був створений. Якщо конструктор був створений явно, то ініціалізації по замовчуванні буде відмінена і повинна здійснюватись конструктором. У випадку, коли конструктор по замовчуванні був створений компілятором (неявний спосіб), то неявно також буде ініціалізація: поля типу POD-клас будуть ініціалізовані по замовчуванні нульовими значеннями, а для полів типу не POD-клас буде викликано конструктор по замовчуванні. Такий підхід використовується для усіх дочірніх об'єктів, які входять до складу полів даного типу.

У випадку використання формату new *A* стандарт *C++11* висуває такі умов:

• якщо тип A є POD-класом, то будь-яка ініціалізації об'єкта відміняється;

• якщо тип A не є POD-класом, то буде викликано конструктор по замовчуванні (явно визначений, або заданий компілятором). У будь-якому випадку жодної автоматичної ініціалізації для полів об'єкта здійснюватися не буде. Для практичного підтвердження сказаного вище розглянемо такий приклад

Приклад № 10.5.

```
#include <iostream>
#include <cstdlib>
using namespace std;
struct A
{
public:
unsigned char n;
};
struct B
{
public:
unsigned char n;
B(){}
};
void main(void)
{
// allocate static memory to place the object A and B
char p[6];
// Fill the memory cell
memset(p, 60, sizeof(p));
p[5] = '\0';
cout << "- string (memset) = " << p ;
// Create an object in memory is clearly filled with number 60
                A* a = new (p) A;
cout << "\n new A: A.n = " << a->n << "\n string
(new A) = " << p << endl;
// Fill the memory cell
memset(p, 109 , sizeof(p));
p[5] = '\0';
cout << "- string (memset) = " << p ;
// Create an object in memory is clearly filled with number 109
A* a1 = new (p) A();
cout << "\n new A(): A.n = " << a1->n << "\n string (new
A()) = " << p << endl;
// Fill the memory cell
memset(p, 60, sizeof(p));
p[5] = '\0';
cout << "- string (memset) = " << p ;
// Create an object in memory is clearly filled with number 60
B* b = new (p) B;
```

```
cout << "\n new B: B.n = " << B->n << "\n string
(new B) = " << p << endl;
// Fill the memory cell
memset(p, 109 , sizeof(p));
p[5] = '\0';
cout << "- string (memset) = " << p ;
// Create an object in memory is clearly filled with number 109
B* b1 = new (p) B();
cout << "\n new B(): B.n = " << b1->n << "\n string (new
B()) = " << p << endl;
}
```

Видно, що у випадку структури A після виконання інструкції new A елемент a.n не був ініціалізований і рівний символу '<', яким були попередньо заповнені комірки. А після виконання інструкції new A() елемент a1.n був нульового значення, тобто відбулась ініціалізація замовчуванні.

У випадку структури B, яка на відміну від структури A , містила явно заданий конструктор, незалежно від інструкції new B чи new B() жодної ініціалізації по замовчуванні не відбулось, Це означає, що ініціалізація повинна бути здійснена явно у конструкторі об'єкта, оскільки по замовчуванні для POD-об'єктів жодних ініціалізаційних дій не здійснюється.

Важливим є те, що для POD-типів у конструкціях з операторам new допускається використання списків ініціаліції, наприклад для типу A з Прикл. № 10.5 допустимою є така ініціалізація

```
A* p = new A{1};
```

Виклики конструкторів визначають принципові відмінність використання операторів new і delete та функцій malloc() (чи calloc()) і free(). Наприклад

```
#include <malloc.h> class D
{
public:
D()
{
cout << "Constructor is running" << endl;
}
~D()
{
cout << "Destructor is running" << endl;
}
};
```

```
...
D* d = new D[5]; delete [] d;
d = (D*)malloc( 5*sizeof(D) ); free (d);
...
```

За цим кодом можна побачити, що виклики конструкторів та деструкторів не відбуваються у випадку використання функцій malloc(), calloc() і free() відповідно. Лише використання операторів new та delete забезпечує коректне звільнення пам'яті. Треба пам'ятати, що оператор delete[] гарантує виклик кондеструкторів для усіх об'єктів масиву, а delete лише для того, на який вказує вказівник-параметр.

При використанні new необхідно строго вказувати об'єм необхідної пам'яті. Цей об'єм на етапі компіляції може бути невідомим, але він повинен бути відомим під час виконання програми. Як вже відзначалось у випадку не виконання запиту виділення пам'яті (по будь-якій причині) оператор new генерує виключення (критичну ситуацію) std::bad_alloc (або похідне від нього). Якщо програма його не перехоплює (не обробляє), то викликається обробник, встановлений функцією set_new_handler(). Це означає, що випадку, коли засобом set_new_ handler() користувацький обробник не встановлено, то програма аварійно завершує своє виконання.

Функція для встановлення користувацькогообробника set_new_handler() визначена у бібліотеці <new> і має такий прототип

```
std::new_handler set_new_handler(std::new_handler new_p)
```

де std::new_handler є псевдоіменем вказівника на функцію

```
typedef void (*new_handler)();
```

За допомогою параметра функція set_new_handler можна встановити новий (користувацький) обробник. При цьому повертається адреса попереднього обробника. Наприклад

Приклад № 10.6.

```
#include <iostream>
#include <new>
std::new_handler old;
void mem_error()
{
std::cerr << "Impossible to satisfy the allocation request\n";
std::set_new_handler( old );
}
```

```
void main(void)
{
old = std::set_new_handler( mem_error );
try
{
while( true )
new int[100000000L];
}
catch( const std::bad_alloc& e )
{
std::cout << e.what() << '\n';
}
```

В наведеному коді, якщо оператор new не зможе виділити пам'ять для розташування 100 000 000 цілих довгих чисел, то буде викликана функція mem_error, яка виведе повідомлення про помилку і поверне попередній обробник. Наступна помилка виділення пам'яті згенерує виключення bad_alloc. Оскільки користувацького обробника вже не існує, то програма завершиться із системним повідомленням про помилку виділення пам'яті.

У практичних задачах користувацький обробник повинен робити хоча б одну із наступних дій:

- Знайти додаткову пам'ять. У результаті наступна спроба виділити пам’ять вже може виконатись успішно. Типовим способом реалізації цього підходу є попереднє виділення пам'яті і звільнення її всередині обробника при виникненні помилки першої спроби виділення.
- Встановити інший обробник. Тут сповідується ідея того, що якщо даний обробник не може забезпечити виділення пам'яті, то це може зробити інший обробник.
- Видалити користувацький обробник інструкцією set_new_handler(nullptr). У цьому випадку при наступному невдалому виділенні буде згенеровано виключення bad_alloc.
- Генерувати виключення bad_alloc, або похідне від нього. Такі виключення не перехоплює оператор new, а тому можна здійснити далекий перехід в іншу область коду, яка може обійти проблему невиділення пам'яті.
- У випадку коли виділення пам'яті є критичним для подальшого виконання коду, завершувати виконання програми, наприклад функціями abort() або exit().

Функція std::new_handler get_new_handler() дає змогу отримати адресу встановленого обробника. Допустимим є нульове значення, що свідчить про відсутність обробника.

У випадку невиконання запиту пам'яті для масиву оператор new згенерує виключення std::bad_array_new_length, яке є похідним від std::bad_ alloc.

У випадку успішного виділення пам'яті розмір блоку можна визначити функцією size_t _msize(void*), яка визначена у бібліотеці malloc.h. Незважаючи на те, що функція була призначена для сумісної роботи із функціями бібліотеки malloc.h, у своїх нинішніх реалізаціях вона успішно працює з операторами мови C++, наприклад

```
long* m = new long;
cout<< _msize(m) << '\n';
```

10.2. Посилання

Змінні-посилання (references) є окремим різновидом вказівників і також містять адресу іншої змінної. За визначенням посилання – є альтернативним іменем об'єкта. Синтаксис оголошення посилань є таким

```
T & ref = object;
T & ref ( object );
```

де object є екземпляром типу T.

Така сучасна мова програмування як Java взагалі використовує посилання взамін вказівників. У C# посилання є основою адресації, в той час як використання вказівників не рекомендований розробником спосіб. Це зумовлено тим, що посилання за своєю суттю є безпечним константим вказівником, але за жодних умов розглядатись вказівником на константу. Наступний приклад найкраще ілюструє ідею посилання

Приклад № 10.7.

```
#include <iostream>
 using namespace std;
void main(void)
{
int i = 1;
int const * const p1 = &i; int& p2 = i;
cout << "Variable address: 0x" << hex << &i << '\n';
cout << "Reference address: 0x" << hex << &p2 << '\n';
cout << "Pointer address: 0x" << hex << &p1 << '\n';
cout << "Pointed address: 0x" << hex << p1 << '\n';
}
```

З результату виконання видно, що адреси змінної i та посилання p2 є однаковими. На відміну від цього адреси посилання p2 та константного вказівника p1 є різними.

Це означає, що незважаючи на те, що посилання та вказівники відносяться до вказівникових типів, синтаксично вони виступають змінними різних типів. Більше того, операція p1 = p2 є недопустимою. Використання явного приведення типів p1 =(int) p2 є також недопустимим, оскільки явна ініціалізація вказівника є забороненою. Використання приведення p1 =(int*) p2 призведе до ініціалізації вказівника p1 значення посилання p2, тобто він адресуватиме комірку пам'яті із адресою 1. Допустимим також є варіант ініціалізації вказівника p1 засобом операції взяття адреси: p1 =&p2. У цьому випадку адресоване вказівником p1 значення буде містити адресу пам'яті, яка була виділена під змінну і і яка адресувалась також посиланням p2.

Зворотна операція p2 = p1 є також недопустимою. Якщо використати будь-яку варіацію операції приведення типу, наприклад p2 = (int)p1, p2 = (int&) p1, або ін., то результатом буде запис нового значення у комірку, яку посилається p2. Це зумовлене тим, що змінити адресу, яка використовується змінною посиланням змінювати заборонено. Компілятор, подібно до випадку звичайної змінної, намагатиметься усіма засобами забезпечити незмінність адреси посилання. Наприклад, у наведеному продовженні коду з 'Прикл. № 10.7

```
int* p3 = new int [10] ;
p3[0] = 3;
//void* p4 = memcpy((const_cast<int*> (&p2)), p3, 1);
void* p4 = memcpy(&i, p3, 4);
cout << "Void Pointed address (p4): " << hex << p4 <<
"\n" "Reference address: " << hex << &p2 <<
"\n" "Pointed address (p3): " << hex << *p3 <<
"\n" "Variable : " << hex << i << "\n";
p4 = memcpy(p4, &p3, 4);
cout << "\nPointed address (p4): " << hex << p4 <<
"\n" "Reference address: " << hex << &p2 <<
"\n" "Variable : " << dec << i << "\n";
p4 = (void*) p3;
cout << "\nPointed address (p4): " << hex << p4 <<
"\n" "Reference address: " << hex << &p2 <<
"\n" "Variable : " << dec << i << "\n";
```

жодна операція, будучи допустимою, не забезпечить зміни значення, яке використовується посилання для доступу до пам'яті.

Посилання повинно бути обов'язково ініціалізоване при визначенні. Це означає, що код створення нової змінної int &i є неправильним. Тому посилання з одного боку можна вважати іменем об'єкта, а з іншого - постійним вказівником. Це дозволяє використовувати посилання у синтаксисі звичайної змінної,

наприклад p2 = 3, або int j = p2. В обидвох випадках код треба трактувати так: *p2=3 і int j = *p2. Але явна операція розіменування посилання є забороненою.

Посилання - це окремий тип. Наприклад змінної p2 з Прикл. № 10.7 є int&. Але, оскільки в більшості операцій мови C++11 змінна p2 буде сумісна з типом int, то оператор динамічної ідентифікації типу typeid(p2).name() поверне тип int. Саме такий тип змінної p2 буде прийматись в неадресних операціях із цією змінною.

Посилання є вказівниковим типом у адресних операціях. Незважаючи на це, посилання не сумісні із вказівниками. На відміну від вказівників:

- посилання обов'язково ініціалізуються при оголошенні, посилання не може містити null-значення;
- посиланню не можна присвоювати нову адресу;
- посилання не займає зайвої пам'яті і може знищуватись раніше самого об'єкта, яке воно адресує;
- не можна створювати масив посилань;
- для посилань є зайвою операція розіменування; ними оперують як звичайними змінними.

Поняття безтипових посилань (void&) не існує.

При створенні посилання можна використовувати зарезервовані слова decltype і auto. Синтаксис цього використання є таким

```
decltype ( (object) ) ref = object;
```

Тут вираз (object) є rvalue. За цим синтаксисом оголошення змінної p2 з Прикл. № 10.7 могло бути таким: decltype ((i)) p2 = i.

У ранніх версіях мови C++ допускалось створення посилання за ініціалізатором типу, який не був адресним виразом. У цьому випадку здійснювалось виділення пам'яті для змінної, ОД якої визначалась за загальними правилами, а її адреса ініціалізувала посилання. Це допускало конструкції виду: int& j=7. У C++11 такі конструкції є недопустимі посилання повинно ініціалізоватись адресами lvalue-ділянок пам'яті. У нинішніх реалізаціях C++ використання літеральних значень допускається лише за умови використання rvalue-посилань. Ці посилання будуть розглядатись нижче у параграфі.

У переважній більшості випадків посилання використовуються трьома способами

- посилання можна передавати у функцію;
- посилання можна повертати з функції. Тут треба бути обережним. Якщо функція повертає посилання, то це повинна бути адреса реально існуючого доступного об'єкта.

Посилання може адресувати константу

```
const int &i = 5;
```

Оголошення посилання з модифікатором const робить неможливою зміну значення, на яке вказує посилання.

Посилання може використовуватись із оператором typedef. Наприклад, typedef int& i; Тепер ідентифікатор i визначає псевдоім'я для типу int&.

Посилання може бути зовнішнім, наприклад extern int& i. Це, фактично, єдиний спосіб оголосити неініціалізоване посилання. Але треба розуміти, що у випадку використання декларації extern визначальним є не створення нової сутності, а організація доступу до вже існуючої.

Наведемо приклади оголошення посилань

Приклад № 10.8.

```
#include <iostream> void f(int) {} struct  A  {}; struct B : A {};
using namespace std;
void main(void)
{
int i = 1;
int& ref1 = i;// посилання на об'єкт n
const int& c_ref(i);// посилання на константу volatile int& v_ref(i);   //
посилання на volatile змінну int& ref2 = ref1;       // ще одне посилання на об'єкт n
//int& bad_ref = c_ref;    // недопустиме посилання
int& ref3 = const_cast<int&>(c_ref); // треба зняти const
void (&f_ref)(int) = f;       // посилання на функцію
int m[3];
int (&m_ref)[3] = m;        // посилання на масив
int& ref4 = *new int;       // посилання на динамічну змінну
int* p = new int[5];
int* (&ref5) = (p);  // посилання на динамічний масив
B b;
A& base_ref = b;    // посилання на частину об'єкта
}
```

У мові C++11 існує ще один тип посилання – rvalue-посилання. Відповідно звичайні посилання тепер називаються lvalue-посиланнями.

Посилання rvalue є виразами для створення анонімних тимчасових об'єктів. Це дає змогу отримати адресацію перш за все анонімних об'єктів, зокрема тих, які знаходяться з правої сторони оператора присвоєння.

Синтаксис оголошення rvalue-посилання є таким

```
T && ref = object;
T && ref ( object )111;
```

Поява подвійного амперсанда '&&' в оголошенні змінної визначає rvalue-посилання. Але в одних випадках ця змінна є rvalue-посилання, а в інших – lvalue-посилання. Таким чином, входження конструкції '&&' у вихідний код може насправді мати сенс одного амперсанда '&', тобто синтаксично є поява rvalue посилання ('&&'), а насправді – lvalue ('&'). Rvalue-посилання можуть адресувати (бути зв'язаними) лише із rvalue-значеннями. А от lvalue-посилання, які цільово адресують lvalue-значення, за певних обставин бути зв'язаними і з rvalue-значеннями. Саме можливість різної трактовки синтаксичної конструкції '&&' і призвела до появи ще одного визначення, а саме універсальні посилання (universal reference) або нормальні посилання.

Відмінність rvalue від lvalue найпростіше пояснити такими міркуваннями:

- якщо можна взяти адресу виразу, то цей вираз є lvalue;
- якщо результат виразу (тип виразу) є посилання (наприклад, T& або const T& і т.д.), то цей вираз є lvalue.
- у всіх інших випадках вираз є rvalue.

На практиці відмінність rvalue від lvalue пояснюється так будь-яка іменоване rvalue-посилання трактується як lvalue. В противному випадку це rvalue.

Концептуально rvalue має стосунок до тимчасових об'єктів, які повертаються функціями, або створюються в результаті неявних операцій приведення типів. Більшість літеральних константи та узагальнених параметрів шаблонів є також rvalue.

Розширення поняття rvalue дало можливість реалізації динамічної операції приведення lvalue до rvalue : static_cast<T&&>(lval). Оператори динамічної ідентифікації типів будуть описані у розділі про RTTI.

Цільовим призначенням rvalue-посилань є реалізація семантики перенесення (move semantics). Основною ідеєю цієї семантики є прискорене створення нових складних об'єктів шляхом перенаправлення вказівників на вже виділену пам'ять. Такий підхід дає змогу уникнути повільних операцій виділення та звільнення пам'яті, знищення тимчасових об'єктів та створення постійних. Типовим прикладом є уникнення використання конструктора копіювання при передаванні/поверненні об'єкта в/з тіло функції. Достатньо лише використати конструктор перенесення, який перенесе вказівник і довжину у новий об'єкт класу.

Використовуючи базові структури з Прикл. № 10.8 можна навести такий приклад оголошення rvalue-посилань

Приклад № 10.9.

```
#include <iostream>
using namespace std; void f()
{
int&& i = 1;  // зв'язування з літералом
```

```
}
strcut A {};
strcut B : A
{
public:
operator int(){ return 1; }
};
B bar() { return B(); }
int j = 0;
int&& rval = j;                 // rvalue, яке насправді є lvalue;
int& lval = j;          // rvalue не може адресувати lvalue;
auto&& rval2 = rval;        // rvalue, яке насправді є lvalue;
auto&& rval3 = lval;        // rvalue, яке насправді є lvalue;
auto&& rval4 = j;                       // rvalue, яке насправді є lvalue;
void main(void)
{
cout << j << ' ' << rval << ' ' << lval << ' ' << rval2 << ' '
<< rval3 << ' ' << rval4 << '\n';
j = 1;
cout << j << ' ' << rval << ' ' << lval << ' ' << rval2 << ' '
<< rval3 << ' ' << rval4 << '\n';
rval3 = 2;
cout << j << ' ' << rval << ' ' << lval << ' ' << rval2 << ' '
<< rval3 << ' ' << rval4 << '\n'; int i = 1;
// int& bad_ref2 = 1;       // помилка організації посилання const int& cref = 1;
// зв'язування lvalue з rvalue int&& rref = 1;         //зв'язування з rvalue
const A& cref2 = bar(); // lvalue-посилання на частину об'єкта A&& rref2 = bar(); // rvalue-посилання на частину об'єкта int&& xref = static_cast<int&&>(i); // зв'язування з lvalue i
//       int&& copy_ref = i; // помилка зв'язування з lvalue double&& copy_ref = i;     // зв'язування з тичасовою змінною B b;
int&& conv_ref = b; // rvalue-посилання на результат приведення
}
```

Аналіз наведеного коду, написаного для компілятора CBGSS44, пропонуємо зробити самостійно.

У додаток до rvalue- та lvalue-посилань мова C++11 ще визначає такі посилання: prvalue, xvalue та glvalue. Для перевірки змінної на тип посилання, або окремо на lvalue чи rvalue, бібліотека <type_traits> має шаблони template< class T > struct is_reference, template< class T > struct is_lvalue_reference та template< class T > struct is_rvalue_reference, наприклад

```
#include <type_traits>
class A {};
void main(void)
{
	std::cout << std::boolalpha <<
	std::is_reference<A>::value << '\n' <<
	std::is_reference<A&>::value << '\n' <<
	std::is_reference<A&&>::value << '\n' <<
	std::is_reference<int>::value << '\n' <<
	std::is_reference<int&>::value << '\n' <<
	std::is_reference<int&&>::value << '\n'<<
	std::is_lvalue_reference<A>::value << '\n' <<
	std::is_lvalue_reference<A&>::value << '\n' <<
	std::is_lvalue_reference<A&&>::value << '\n' <<
	std::is_lvalue_reference<int>::value << '\n' <<
	std::is_lvalue_reference<int&>::value << '\n' <<
	std::is_lvalue_reference<int&&>::value << '\n'<<
	std::is_rvalue_reference<A>::value << '\n' <<
	std::is_rvalue_reference<A&>::value << '\n' <<
	std::is_rvalue_reference<A&&>::value << '\n' <<
	std::is_rvalue_reference<int>::value << '\n' <<
	std::is_rvalue_reference<int&>::value << '\n' <<
	std::is_rvalue_reference<int&&>::value << '\n';
}
```

10.3. Масиви і вказівники

Масивом називається скінченна множина (набір) однотипних елементів, які розміщуються в пам'яті послідовно. Визначальною для масивів є операція індексації, яка забезпечує окремий доступ до елементів за індексом. Результатом індексації є посилання на визначену індексом комірку пам'яті масиву. Відповідно до цього масив є логічною структурою із довільним доступом.

Основними перевагами масиву є:

- зручність і простота визначення адреси елемента за індексом;
- однаковий час доступу до будь-якого елемента.

Розмірність масиву – це кількість індексів, які необхідні для отримання посилання на комірку пам'яті.

У загальному випадку розглядають такі масиви:

- статичні – масиви, розміри яких є преозначеними і незмінними;
- динамічні – масиви, розміри яких є можуть змінюватись в процесі виконання програми;
- асоціативні – масиви, доступ до елементів яких здійснюється за допомогою хеш-функцій.

- гетерогенні – масиви, елементи яких є різних типів.

Допускаються несуперечливі комбінації масивів, наприклад динамічний асоціативний.

Мова C++ підтримує лише перші два види масивів. Розглянемо їх детальніше.

Статичні масиви у C++11 можна створювати двояко. Перший спосіб полягає у використанні константного вказівника, а другий використання шаблону array. Другий спосіб детальніше буде розглядатись у p.

Синтаксичні конструкції створення масивів за допомогою константного вказівника найкраще продемонструвати на прикладах

```
// створення масиву: розмірність (задана явно) - 10 елементів;
//ініціалізація повна по замовчуванні
int ari1[10];
const int Size() { return 1; }
// int ari1_1[Size()]; - помилка - не константою задана розмірність
// створення масиву: розмірність (задана явно) - 10 елементів;
//ініціалізація повна по замовчуванні
#define SZ 10 int ari1_2[SZ];
// створення масиву: розмірність (задана явно) - 10 елементів;
//ініціалізація повна по замовчуванні
const int sz = 10; int ari1_3[sz];
// створення масиву: розмірність (задана неявно) - 3 елементи;
//ініціалізація повна і явна
int ari2[] = {1,2,3};
// створення масиву: розмірність (задана неявно) - 3 елементи;
//ініціалізація повна і явна
int ari2_1[] = {1,2,3,};
// створення масиву: розмірність (задана явно) - 3 елементи;
//ініціалізація повна і явна
int ari3[3] = {1,2,3};
// створення масиву: розмірність (задана явно) - 3 елементи;
//ініціалізація повна і явна
int ari3_1[3] = {1,2,3,};
// створення масиву: розмірність (задана явно) - 3 елементи;
//ініціалізація часткова і явна
int ari4[3] = {1,2};
// створення масиву: розмірність (задана явно) - 3 елементи;
//ініціалізація часткова і явна
int ari4_1[3] = {1,2,};
```

Коментарі, наведені у коді, пояснюють особливість кожної конструкції. Для полегшення розуміння поняття масиву на Рис. 10.1 наведено схему фізичного розміщення масиву (на прикладі ari2) в пам'яті.

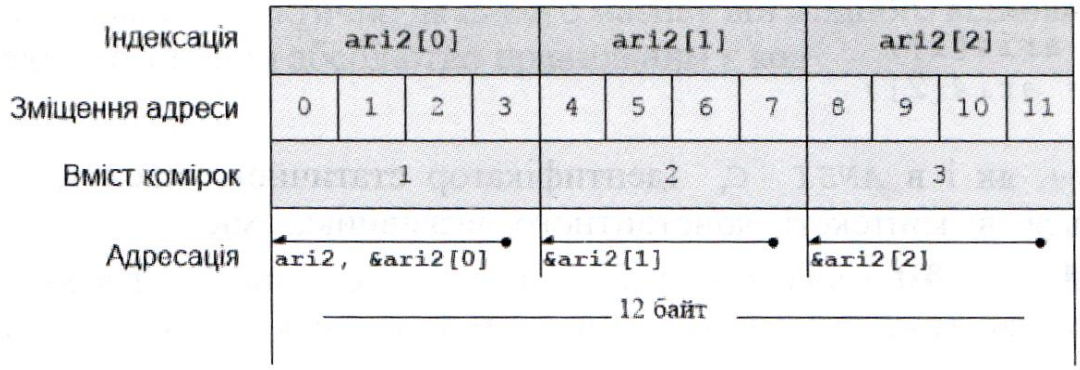


Рис. 10.1. Схема розміщення масиву ari2

Зазначимо, що

- пам'ять під статичний масив виділяється автоматично в ділянці, яка визначається ОД ідентифікатора масиву. Звільнення пам'ять також відбувається автоматично в залежності від ОД.
- типом елементів масиву може бути будь-який допустимий у мові C++11 тип;
- у випадку вказання ініціалізації по замовчуванні мається на увазі глобальний масив. Для локального масиву ініціалізація по замовчуванні є відсутня. Тому у загальному випадку, якщо відсутній явно заданий ініціалізатор, масив слід сприймати невизначеним;
- у випадку часткової ініціалізації, решта елементів масиву будуть ініціалізовані за правилом, яке визначене у попередньому пункті;
- розмірність масиву визначається цілочисельною додатною константою. Тому замість оголошення int ari0[3.14]; треба писати так int ari0[(int)3.14];.
- Створення масиву ari1_1, є помилковим лише у випадку, коли є ari1_1 є глобальним масивом. Для локальних масивів допускається визначення індексів результатом, який повертається іншими функціями. При цьому тип результату не обов'язково повинен бути константним – достатньо, щоб значення було цілочисельним. Наприклад

```
const int Size() { return 1; }
void main(void)
{
int ari0[Size()];
}
```

Розмірність масиву в межах ОД можна легко визначити за таким кодом

```
sizeof(ari0) / sizeof(int)
```

або, в залежності від типу елементів масиву

```
sizeof(ari0) / sizeof(ari0[0]).
```

або з використанням функції _msize(void*),

```
#include <malloc.h>
_msize(ari0) / sizeof(ari0[0]).
```

При виході за межі ОД, наприклад при передаванні масиву в тіло функції, інформація про розмір масиву втрачається (передається виключно вказівник). Тому розмірність масиву не може бути визначеною.

Індексація елементів масиву у мові C/C++ завжди починається із нуля. Для доступу до елемента масиву використовується оператор індексації, який у загальному випадку має вид R T::operator [](S index), де R, T, S є типами мови C++.

У випадку використання вбудованого (не перевантаженого користувачем) оператора індексації буде повернуто посилання на визначений індексом елемент масиву, тому допустимими є такі записи

```
ari4[2] = 4;
int i = ari4[2];
i =      ari4[2] = 7;
int& r = ari4[1];
int& rr = ari4[2];
```

У C++, як і в ANSI C, ідентифікатор статичного масиву, який завжди розглядається в контексті константного вказівника, містить адресу початку розміщення у пам'яті.

```
char ach[30]; // тут ach є адресою &ach[0];
```

Це робить простою організацію сторонніх вказівників на масиви даних

```
int ari[10]; int * j;
// перший варіант організації вказівника j на масив achArray
j = &ari[0];
// другий варіант організації вказівника j на масив achArray
j = ari;
```

Наведений код має значне ідеологічне значення – незважаючи на те, що масив у C++ розглядається в контексті окремого типу даних, насправді він є звичайним вказівником.

Одним із результатів типізації масиву є те, що змінні int i[5] і int j[5]. вважаються різного типу і це незважаючи на те, що насправді вони є константними вказівниками.

Оператор індексації відіграє роль оператора доступу для такого типу як масив. Визначення адреси елемента масиву здійснюється шляхом пересування вказівника відносно початку на кількість байтів, яка визначається добутком значення індексу на розмір типу. Проілюструємо це на організації доступу до елементів масиву через сторонній вказівник. Цей процес складається з двох етапів:

- ініціалізація вказівника адресою першого чи останнього елемента масиву;
- використання циклічного оператора для доступу до елементів масива або маніпуляції адресою, яку містить вказівник. Зокрема, якщо j = &ari[0], то наступні звертання до i-го елемента масиву ari є рівносильними

```
ari[i] = j[i] = *(j + i);
```

Зауважимо: хоча операції * і & мають найвищий пріоритет, проте код *j++ поверне посилання на елемент масиву з індексом збільшеним на одиницю (читається даний вираз справа наліво). Тобто, при умові j = &ari[0] маємо

```
*j++ = ari[1] = *(j + 1) = *(j++).
```

Для того щоб збільшити на одиницю значення елемента масиву, треба забезпечити найпершим виконання операціях розіменування: (*j)++.

Основною проблемою оператора індексації, є те, що насправді його основний функціонал призначений для пересування поточного вказівника з метою отримання посилання. Фактично до самого масиву він жодного відношення немає. Тому, з точки зору компіляції, абсолютно правильним є код

```
int ari[2] = 4;
ari4[5] = 6;
```

Проте результат виконання є непередбаченим. Помилкова спроба доступу до елемента з індексом який рівний значенню розмірності навіть отримала власну назву "помилка останнього стовпа" (fens post error). Найкращим з можливих результатів є генерування виключення на періоду виконання. Це, принаймні, явно засвідчить існування помилкового коду.

Статичні масиви можуть бути багатовимірними, зокрема двовимірними, наприклад

Приклад № 10.10.

```
#include <iostream>
using namespace std;
// створення масиву: розмірність (задана явно) - 2x3 елементи;
//ініціалізація повна по замовчуванні
#define SZ 2
int ari0[SZ][3];
// створення масиву: розмірність (задана явно) - 3x2 елементи;
//ініціалізація повна по замовчуванні
const int sz = 3; int ari0_1[sz][SZ];
// створення масиву: розмірність (задана явно) - 2x3;
//ініціалізація повна по замовчуванні
int ari1[2][3];
// створення масиву: розмірність (задана явно) - 2x3;
//ініціалізація повна і явна
int ari2[2][3] = {1, 2, 3, 4, 5, 6};
// створення масиву: розмірність (задана явно) - 2x3;
//ініціалізація повна і явна
int ari2_1[2][3] = { {1, 2, 3}, {4, 5, 6} };
// створення масиву: розмірність (задана неявно) - 2x3;
//ініціалізація повна і явна
int ari3[][3] = { 1, 2, 3, 4, 5, 6 };
// створення масиву: розмірність (задана неявно) - 2x3;
//ініціалізація повна і явна
int ari3_1[][3] = { {1, 2, 3}, {4, 5, 6} };
// створення масиву: розмірність (задана неявно) - 2x3;
//ініціалізація часткова і явна
int ari3_2[][3] = { {1, 2}, {4, 5} };
// створення масиву: розмірність (задана неявно) - 2x3;
//ініціалізація часткова і явна
int ari3_3[][3] = { 1, 2, 3, 4 };
// створення масиву: розмірність (задана неявно) - 2x3;
//ініціалізація часткова і явна
int ari3_4[][3] = { 1, {2, } };
// створення масиву: розмірність (задана неявно) - 2x3;
//ініціалізація часткова і явна
int ari3_5[][3] = { {1, 2 }, 4 };
//int ari4[2][] = { 1, 2, 3, 5, 6, 7 }; помилка
void output (char* name, int m[][3] )
{
cout << name << '\n';
```

```
for (int i = 0; i < 2; i++) for (int j = 0; j < 3; j++)
cout << " (" << i << ',' << j << ") - adr.: 0x" << &m[i][j]
<< " - val.: " <<       m[i][j] << '\n';
}
void main(void)
{
output( "ari1", ari1 ); output( "ari2", ari2 ); output( "ari2_1", ari2_1 ); output( "ari3", ari3 ); output( "ari3_1", ari3_1 ); output( "ari3_2", ari3_2 ); output( "ari3_3", ari3_3 ); output( "ari3_4", ari3_4 );
//output("ari3_5", ari3_5, 2); int ari6[2][3];
output("ari6", ari6 );
}
```

Подібно до Рис. 10.1 на Рис. 10.2 наведено фізичне розміщення двовимірного масиву. Доступ до елементів масиву, тобто отримання посилання на відповідну комірку пам'яті, здійснюється подвійним використанням операції індексації, наприклад ari2[0][1] = 7.

Тип масив, кількість розмірностей та елементів у багатовимірних масивах можна визначити шляхом використання шаблонів template< class T> struct *is_array*, template< class *T*> struct *rank* і template< class *T,* unsigned *N* =0> struct *extend* бібліотеки <type_traits>, наприклад

```
#include <type_traits>
...
class A {};
std::cout << std::boolalpha
<< std::is_array<A>::value < '\n'
<<      std::is_array<A[3]>::val
<< '\n'
<<      std::is_array<float>::val
<< '\n'
<<      std::is_array<int>::val
<< '\n'
<< std::is_array<int[3]>::val
<< '\n'
std::cout << std::rank<int[1][ [3]>::value << '\n'
<<      std::rank<int[][2][
[4]>::value << '\n'
<<      std::rank<int>::value       < '\n';
std::cout      <
std::extent<int[3]>::value          < '\n'
<<      std::extent<int[ [4]>::value << '\n'
```

```
<<      std::extent<int[ [4],1>::value << '\n'
<<      std::extent<int[ [4],2>::value << '\n';
<<      std::extent<int[]>::val
<< '\n';
```

У цій же бібліотеці існують засоби для зміни розмірностей масивів, зокрема це шаблони template< class *T*> struct *remove_extent* і template<class *T*> struct *remove_all_extent*. Іншим видом масиву є динамічні масиви. Принциповими їх відмінностями від статичних масивів є те, що організація динамічних масивів здійснюється за допомогою звичайного (неконстантного) вказівника. Це дає можливість користувацького керування процесами створення/знищення масивів. Наприклад

```
#include <malloc.h>
#include <new>
...
int* m = new ( std::nothrow ) int[10];
if (!m) exit(0);
delete [] m;
        m = (int*)malloc( 10*sizeof(int) );
if (!m) exit(0);
free(m);
if ( (m = (int*)calloc113( 10, sizeof(int) ) ) == NULL )
exit(1);
free (m);
volatile char *m1 = new volatile char[20];
if (!m1) exit(0);
        delete m1;
```

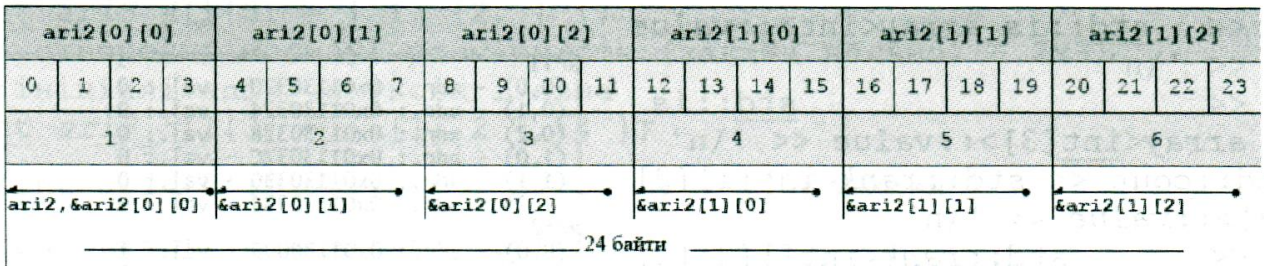


Рис. 10.2. Схема розміщення двовимірного масиву ari2 з Прикл. № 10.10

Використання динамічних масивів дає можливість динамічної зміни їх розміру за допомогою функції void* realloc(void*, size_t), наприклад

```
#include <malloc.h>
```

```
...
long* m = (long*)malloc(10*sizeof(long));
std::cout << _msize(m) / sizeof(m[0])<< '\n';
m =    (long*)realloc(m, 14*sizeof(m[0]) );
std::cout << _msize(m) / sizeof(m[0])<< '\n';
free (m);
if ( ( m = new(std::nothrow) long[3]) != NULL )
{
std::cout << _msize(m) / sizeof(m[0])<< '\n';
m = (long*)realloc(m, sizeof(m[0]) );
std::cout << _msize(m) / sizeof(m[0])<< '\n';
delete [] m;
}
```

Функція realloc може змінити розмір пам'яті будь-якого блоку пам'яті, адреса якого передається функції через параметр. Тому синтаксично правильною є така конструкція

```
int m2[3];
int* m3 = (int*)realloc (m2, 4*sizeof(m2[0]) );
```

Але результатом наведеного коду на етапі виконання буде виключення, оскільки відбулась спроба змінити розмір блоку пам'яті не в області динамічної купи.

Організація динамічних багатовимірних масивів здійснюється подібно до *ANSI C*, наприклад

Приклад № 10.10.

```
#include <iostream>
const unsigned int row = 2;
const unsigned int col = 3;
void main(void)
{
int **m; // оголошення вказівника
m = new int * [row]; // створення матриці
std::cout << "m: Adress - 0x" << &m << " Value - 0x" << m
<< "\n Size:" << sizeof(m) << " Pointed memory size:"
<<_msize(m) <<'\n' ;
for (int i = 0; i < row; i++)
{
m[i] = new int [col];
new int [col]; // розрив у пам'яті
```

```
std::cout << "m[" << i <<"]: Adress - 0x" << &m[i]
<< " Value - 0x" << m[i] << "\n Size:" << sizeof( m[i] )
<< " Pointed memory size:" <<_msize( m[i] ) << '\n';
for (int j = 0; j < col; j++)
{
m[i][j] = i + j; //ініціалізація масиву елементів
std::cout << "m[" << i <<"][" << j <<"]: Adress - 0x" <<
&m[i][j] << " Value - " << m[i][j] <<
" Size:" << sizeof( m[i][j] ) << '\n';
}
}
}
```

Як видно з рисунків в основі створення динамічної матриці *m* лежить побудова масиву вказівників. Кожен елемент *m*[*i*] масиву *m*, у свою чергу, вказує на масив з трьох елементів типу int. Така конструкція не є ефективною з точки зору витрат пам'яті та проведення блокових операцій у пам'яті (копіюванні, зміни розміру і ін.). Проте володіє можливостями динамічного створення та знищення. Більше цього, за допомогою, наприклад двовимірного вказівника, можна створювати матриці не обов'язково прямокутної форми. Для того, щоб знівелювати недоліки та залишити переваги, багатовимірні масиви, розміщують в пам'яті одним блоком і створюють відповідний інтерфейс.

Наприклад для матриці цей код міг би виглядати так

```
const unsigned int row = 2; const unsigned int col = 3;
int& at( int* m, unsigned int i, unsigned int j )
{
return m[ i*col + j ];
}
void main(void)
{
int* m = int [col*row];
for (int i = 0; i < row; i++) for (int j = 0; j < col; j++)
at(m,i,j) = i*col + j; //ініціалізація елементів
}
```

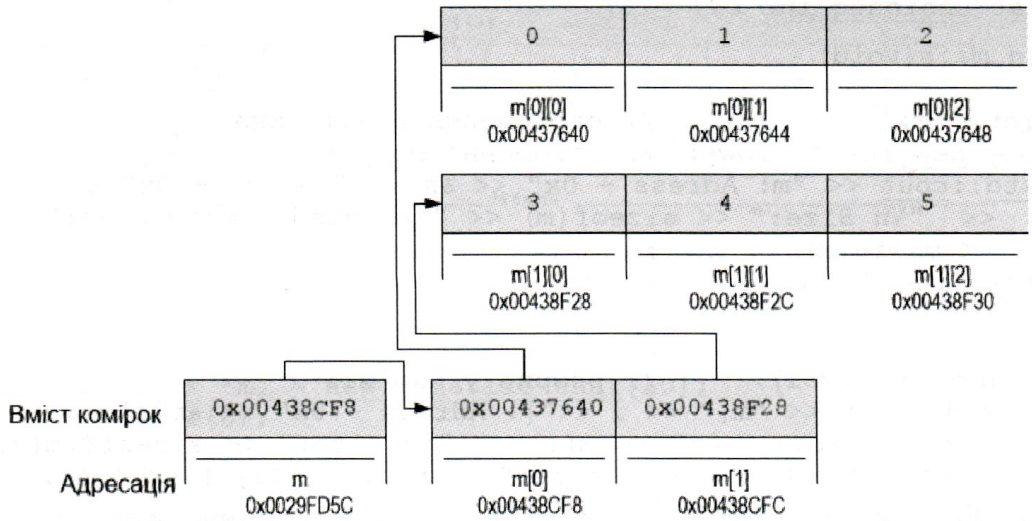


Рис. 10.3. Схема розміщення в пам'яті масиву m з програми наведеної у Прикл. № 10.11

Наостанок наведеному приклад, який ілюструє зв'язок статичної матриці і подвійного вказівника. У цьому прикладі доступ до матриці (двомірного масиву) організовується як масив вказівників на рядки, або як вказівник на вказівник. Аналіз роботи пропонується зробити самостійно.

Приклад № 10.12.

```
#include <iostream>
const unsigned int row = 10;
const unsigned int col = 10;
void main(void)
{
float arfl[row][col];          //оголошення матриці
float * p1[row];     //оголошення масиву вказівників
float ** p2 = p1;    //оголошення подвійного вказівника
int i, j;//оголошенні лічильників циклів
for (i = 0; i < row; i++)      //ініціалізація масиву елементів
{
for (j = 0; j < col; j++)
arfl[i][j] = float(i + j); // матриці
p1[i] = &arfl[i][0]; // вказівників
}
float Suma = 0,Suma_p1 = 0,Suma_p2 = 0;
for (i = 0; i < row; i++) // обчислення суми елементів масиву
for (j = 0; j < col; j++)
{
Suma += arfl[i][j]; // через ідентифікатор матриці
Suma_p1 += *( *(p1 + i) + j); // через масив вказівників
```

```
Suma_p2 += *( *(p2 + i) + j); // через подвійний вказівник
}
std::cout << "Сума елементів матриці порахована через:" << "\n"
<< " - ідентифікатор масиву: " << Suma << "\n"
<< " - подвійний вказівник: " << Suma_p2 << "\n"
<< " - масив вказівників: " << Suma_p1 << "\n";
}
```

Використання вказівників з складеними програмними конструкціями є основним в технології ООП. Усі сучасні професійні бібліотеки розробки прикладних програм, такі наприклад як MFC, VCL і ін., є суцільним нагромадження вказівників. Тому важливим є сумісне використання вказівників і структурованих змінних. Мова C++ допускає таке сумісне використання без жодних обмежень, наприклад

```
struct S
{
int i; int mas[10];
};
...
S * p = new S; S m[10];
S * p2 = m;
for(int j = 0; j < 10; j++)
{
p->mas[j] = j;
p2[j]->i = j;
}
```

Подібно до структур організовуються вказівники на об'єднання.

Лекція 11. Функції. Ідентифікатор функції. Параметри функції та повертання значень функції. Види функцій.

11.1. Поняття функції

Намир Шаммас у своїй книзі назвав функції двигунами програми. І це, очевидно, є так. Будь-яка C++-програма є функцією і будь-яка функція за своєю структурою повторює основну програму.

Функції C++ є аналогами підпрограм і процедур, які використовуються в інших мовах програмування. Вони є зовнішніми по замовчуванні, а тому явне задання класу пам'яті extern є зайвим. Оголошення функцій статичними робить неможливим їх використання у інших модулях програми.

Усі функції C++ мають: ім'я (ідентифікатор функції), тип значення, яке вона повертає за допомогою оператора return (операторів return може бути декілька) і необов'язковий список параметрів.

Функції, які повертають значення, зобов'язані принаймні один раз використовувати оператор return. Функції оголошені як void не повертають значення.

Загальна структура синтаксису оголошення функції має вид:

```
type Name(parametrs)
{
декларації; оператори;
return значення/вираз;
}
```

У мові C++ функція обов'язково оголошується через прототип до моменту першого виклику. Сама реалізація функції може бути в довільному місці. Наприклад

```
int f( char, char );   // короткий прототип
// int f( char a, char b );   // повний прототип
void main(void)
{
int i = f( 'a', 'b' );
}
int f( char a, char b )        // визначення функція
{...;}
```

У наведеному прикладі є дві функції: main() та f(). Функція main() є специфічною функцією. Її специфіка полягає у тому, що ця функція є точкою входу в тіло основної програми. За допомогою цієї функції і типу

розроблюваного програмного рішення, компілятор визначає так звану стартову функцію (start-up function), яка, у свою чергу, є функцією основного потоку (main thread) процесу, що буде породжений при запуску програми на виконання. Виклик функції main() є преозначений створенням процесу. Це визначає автоматичним запуск на виконання функції. Таке неявне використання є основною причиною того, що у C++ функцію main() є завжди один і він є визначеним.

Функція f() використовується в тілі функції main(). Тому обов'язковим є попереднє оголошення функціях за допомогою прототипу. Прототип можна опустити, якщо функція є повністю визначена (разом з тілом) до моменту першого виклику.

У контексті прототипу розрізняють декларацію функції (прототип) та визначення функція. Визначення функціях складається із заголовку (оголошення функції) та її тіла.

Прототип функції є скороченою декларацією функцією. У більшості випадків він складається лише із сигнатури функції. Сама сигнатура, у свою чергу, складається із типу повернення та типу і кількості параметрів. Ідентифікатори параметрів до складу сигнатури не входять. Тут треба зазначити, що у загальному випадку розрізняють довгу і коротку сигнатури. У цьому контексті наведене означення визначає довгу сигнатуру, або просто сигнатуру. Коротку сигнатуру складають лише тип та кількість параметрів.

У тілі будь-якої функції можна оголошувати локальні константи і змінні. Їх ОД обмежується батьківською функцією. Пам'ять, яка виділяється для зберігання значень цих змінних називається стековою або просто стеком. Стек це спеціальна ділянка пам'яті в адресному просторі процесу, яка виділяється для розміщення локальних змінних, необхідних для виконання кожної функції. Сама назва "стек" цієї ділянки походить від принципу роботи з цією пам’яті, яка полягає у використанні концепції логічної структури "стек": останнім надійшов, першим вибув. При розміщенні даних у стек (операція – push), останній послідовно збільшується, а вибиранні (виштовхуванні, операція – pop) – стек зменшується. Вибирання даних є також послідовною операцією, але із зворотним до операції розміщення порядком.

Існує два типи локальних змінних: автоматичні (по замовчуванні) і статичні. Параметри функції відносять до автоматичних локальних змінних.

Автоматичні локальні змінні створюються системою на початку виконання функції, зберігаються у стеку під час її виконання і знищуються по закінченню виконання. Статичні змінні ж зберігаються у глобальній області, тобто є подібними до глобальних. Але на відміну від останніх доступ до статичних локальної є строго детермінованим.

На відміну від автоматичних статичні локальні змінні зберігають свої значення між викликами функції. Якщо статичній локальній змінній присвоїти значення при оголошення, то ця ініціалізація дасть можливість батьківській функції визначити чи виконується вона вперше.

Наведемо приклад використання статичних локальних змінних

Приклад № 11.1.

```
#include <iostream>
using namespace std;
#define m0 "\taddr."
#define m1 "\t\taddr." int i = 10;
int f()
{
int j = 0;
j++;
static unsigned int cnt = 0;
cnt++;
::i++;
cout << " j = " << j << m1 << &j << '\n'
<< " cnt = " << cnt << m0 << &cnt << '\n' ; return cnt;
}
void main(void)
{
int i = 15;
cout << " ::i = " << ::i << m0 << &::i << "\n i = " << i << m1
<< &i << "\n#" << f() << " - call" << endl;
cout << " ::i = " << ::i << m0 << &::i << "\n i = " << i << m1
<< &i << "\n#" << f() << " - call" << endl;
i++;    ::i++;
cout << " ::i = " << ::i << m0 << &::i << "\n i = " << i << m1
<< &i << "\n#" << f() << " - call" << endl;
}
```

Як можна побачити з виконання програми змінні cnt та ::i знаходяться у ділянці пам'яті глобальних змінних. Відповідно усі зміни значень цих змінних, які здійснюються в тілі функції, мають глобальну дію. Змінні j та i знаходяться у стеках відповідних функцій.

Принциповою відмінністю змінної cnt від ::i те що ідентифікатор cnt має локальну ОД. Це означ що використання цього є можливим виключно у тілі функції. Очевидно, що ділянка пам'яті, яка виділялась під змінну cnt є доступною і за межами тіла функції.

11.2. Ідентифікатор функції

Ідентифікатор функції є звичайним глобальним ідентифікатором, на який розповсюджуються правила області видимості. Його цільовим призначення є забезпечення виклику функції, тобто виконання коду, який зосереджений у тілі функції.

Виклик функції у вихідному коді на етапі виконання спричинює таку послідовність дій:

1) організація стеку;
2) перехід по мітці в області коду (перехід на код тіла);
3) виконання кода тіла;
4) повернення значення/очистка стеку;
5) перехід по мітці в області коду (повернення по адресі на наступну команду від точки початкового переходу – зворотна адреса або адреса повернення).

Процедура організації стеку полягає у побудові і заповненні певної структури, подібної до наведеної на 'Прикл. № 11.4, в області стекової пам'яті. Усі дані в області стеку мають послідовні адреси. Перший з них визначає верхню границю стеку. Відповідно до стекової пам'яті поточної функції відноситься та пам'ять, адреси якої лежать нижче значення верхньої границі і до значення границі заповнення стеку. Пам'ять з адресами вище значення вершини, до стеку функції не відносяться.

Наведений порядок і структура стеку є дуже загальними. У кожному конкретному випадку вони можуть частково відрізнятись в залежності від операційного середовища, компілятора та способу виклику функції. Наприклад, на Прикл. № 11.1 наведена конкретна реалізація стеку для компілятора BS5. З наведеної на Прикл. № 11.4 чи Прикл. № 11.1 є послідовностей видно, виконання основного функціоналу, зосередженого у тілі, забезпечується переходом по мітці (адресі) в області коду. Це означає, що ідентифікатор функції є самим звичайним вказівником, тобто адресою з якої починається виконання коду тіла функції. У відповідності до цього тип ідентифікатора функції є вказівниковим. Але це є специфічний вказівник. Його повний тип як вказівника визначається сигнатурою функції. А основна специфіка полягає у тому, що по-перше це є константний вказівник, а по-друге – це вказівник не у область даних, а в область коду.

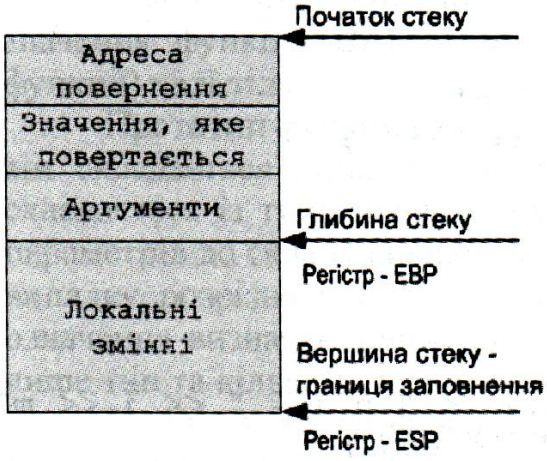


Рис. 11.1. Схема структури функціонального стеку

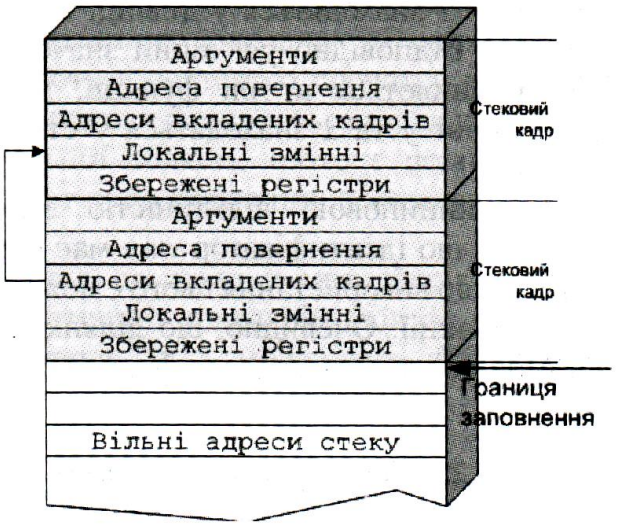


Рис. 11.2. Структури функціонального стеку компілятора BS5

У якості прикладу організації стекової структури наведемо такий приклад

Приклад № 11.2.

```
#include <iostream>
using namespace std;
#define m0 l0 l1
#define m1 l0 l2
#define m2 l9 l2
#define m3 l9 l1
#define m4 h << m2
#define m5 h << m3
#define m6 h << m1
#define m7 h << m0
#define m8 h << l3
#define m9 " reg. val. "
#define ic h << l4 l6
#define dc h << l5 l6
#define l0 "  F"
#define l1 " - (local param. "
#define l2 " - (difference:"
#define l3 "  F - ( reg. var. "
#define l4 "  F - ( Decrement &a:"
#define l5 "  F - ( Increment &a:"
#define l6 "): addr. "
#define l7 ": val. "
#define l8 "  val. "
#define l9 "Main"
#define h hex
```

```
#define d dec
#define GB "Global param. (" typedef unsigned long UL;
UL vEBP, vESP;
int f(int a)
{
int* p = &a;
UL fEBP, fESP;
    asm
{
mov fESP, ESP mov fEBP, EBP
}
cout << m7 "a" l6 << &a << l8 << a << '\n'
<< m7 "p" l6 << &p << l8 << p << '\n'
<< m8 << "EBP" l6 << &fEBP << m9 << fEBP << '\n'
<< m8 << "ESP" l6 << &fESP << m9 << fESP << '\n'
<< m6 "       &a - EBP)" l7 << d << (UL)&a - fEBP << '\n'
<< m6 "       &a - p)" l7 << d << (UL)&a - (UL)&p << '\n'
<< m6 "       &a - varEBP)" l7 << d << (UL)&a - (UL)&fEBP << '\n'
<< m6 "       &a - varESP)" l7 << d << (UL)&a - (UL)&fESP << '\n'
<< m6 "       &a - ESP)" l7 << d << (UL)&a - fESP << '\n'
<< m6 "       EBP - p)" l7 << d << fEBP - (UL)&p << '\n'
<< m6 " EBP - varEBP)" l7 << d << fEBP - (UL)&fEBP << '\n'
<< m6 " EBP - varESP)" l7 << d << fEBP - (UL)&fESP << '\n'
<< m6 "       EBP - ESP)" l7 << d << fEBP - fESP << '\n'
<< m6 "       p - varEBP)" l7 << d << (UL)&p - (UL)&fEBP << '\n'
<< m6 "       p - varESP)" l7 << d << (UL)&p - (UL)&fESP << '\n'
<< m6 "       p - ESP)" l7 << d << (UL)&p - fESP << '\n'
<< m6 "varEBP-varESP)" l7 << d << (UL)&fEBP - (UL)&fESP << '\n'
<< m6 " varEBP - ESP)" l7 << d << (UL)&fEBP - fESP << '\n'
<< m6 " varESP - ESP)" l7 << d << (UL)&fESP - fESP << '\n'
<< ic << (UL)( (&a) - 1 ) << l8 << *( (&a) - 1 ) << '\n'
<< dc << (UL)( (&a) + 1 ) << l8 << *( (&a) + 1 ) << '\n'
<< ic << (UL)( (&a) - 2 ) << l8 << *( (&a) - 2 ) << '\n'
<< dc << (UL)( (&a) + 2 ) << l8 << *( (&a) + 2 ) << '\n';
return *p;
}
int main(int argc, _TCHAR* argv[])
{
int x = 1;
    asm
{
```

```
mov vESP, ESP mov vEBP, EBP
}
cout << h << GB "vEBP" l6 << &vEBP << '\n'
<< h << GB "vESP): addr. " << &vESP << '\n'
<< m5 "argv" l6 << &argv << '\n'
<< m5 "argc" l6 << &argc << " val. " << argc << '\n'
<< h << l9 " - (        register EBP):        val. " << vEBP << '\n'
<< m5 "        x" l6 << &x << " val. " << x << '\n'
<< h << l9 " - (        register ESP):        val. " << vESP << '\n'
<< m4 " &argv - &argc)" l7 << d << (UL)&argv - (UL)&argc << '\n'
<< m4 "        &argv - EBP)" l7 << d << (UL)&argv - vEBP << '\n'
<< m4 "        &argv - &x)" l7 << d << (UL)&argv - (UL)&x << '\n'
<< m4 "        &argv - ESP)" l7 << d << (UL)&argv - vESP << '\n'
<< m4 "        &argc - EBP)" l7 << d << (UL)&argc - vEBP << '\n'
<< m4 "        &argc - &x)" l7 << d << (UL)&argc - (UL)&x << '\n'
<< m4 "        &argc - ESP)" l7 << d << (UL)&argc - vESP << '\n'
<< m4 "        EBP - &x)" l7 << d << vEBP - (UL)&x << '\n'
<< m4 "        EBP - ESP)" l7 << d << vEBP - vESP << '\n'
<< m4 "        &x - ESP)" l7 << d << (UL)&x - vESP << '\n'
<< h << l9 " - function addr. :" <<
(reinterpret_cast<int*> (f)) <<'\n' ;// адреса з f f(2);
return 0;
}
```

Використаний у програмі регістр ESP вказує на вершину стеку, тобто адресу, за якою буде заноситись командою push наступна змінна. Регістр EBP визначає так звану глибину стеку, тобто містить адресу, починаючи з якої у стек заносяться або вибираються дані. Параметри функції мають додатній зсув відносно значення EBP, локальні змінні – від'ємний. Треба пам'ятати, що 32-х розрядні регістри EBP і ESP є регістрами зсуву. По адреса буде визначатись парою регістрів SS : EBP, де SS – 16-и розрядний регістр , який визначає сегмент стеку. Зчитування значення регістрів здійснювалось вставкою асемблерного коду, яка визначалась службовим словом asm.

У наведеному коді варто звернути увагу на використання ідентифікатора функції f, як звичайної змінної.

11.3. Параметри функції та повертання значень функціями

У C++ інформація про тип та кількість параметрів "розміщується" в імені функції. Такий підхід називається безпечним зв'язуванням функції (type-safe linkage).

Використання безпечного зв'язування забезпечує

- контроль типів;

- виявлення відмінностей в оголошеннях функцій;
- реалізацію механізму перевантаження.

Типовим прикладом використання безпечного зв'язування є такий код

– файл 1.c (прототип)

```
int f (unsigned int);
```

– файл 2.c (визначення функції)

```
int f (int){}
```

За принципом безпечного зв'язування функція, оголошена у файлі 1.c, буде вважатись невизначеною. А функція, визначена у файлі 2.c, такою, яка не має прототипу.

Функція може не приймати параметрів. Порожній список параметрів в C++ оголошується за допомогою двох форматів: оголошення порожнім списку або використання зарезервованого слово void, для явного вказування цього, наприклад

```
int f();
int f(void);
```

Такими оголошеннями компілятору дається інформація, що кількість параметрів рівна нулю. Тут треба зазначити, що у мові C формат int f(); інтерпретувався як невизначена кількість параметрів. У цьому випадку не здійснюється жодна перевірка типів. Тому можливі будь-які виклики функції, наприклад f(10);, f(10, 'c');.

Список параметрів може містити один чи декілька параметрів, які, відповідають аргументам даної функції при її фактичному виклику. Кожен параметр повинен мати власний тип. Неможна використовувати один тип для оголошення декількох параметрів.

Існує три методи передавання параметрів функції:

- за значенням (класичний метод, коли копія змінної через стек передається у тіло функції, а тому можливе виключення переповнення стеку. У даному випадку функція не може змінити вихідне значення), наприклад

```
...
int f(int i)
{
i = 4;
return i;
}
void main(void)
```

```
{
int j = 10;
std::cout << "j = " << j << '\n'; std::cout << "f = "<< f(j) << '\n'; std::cout << "new j = " << j << '\n';
}
```

За наведеним кодом змінна j зберегла своє значення між викликами функції, незважаючи на спробу змінити її значення у тілі функції. Зміна значення стосувалась лише стекової змінної i, яка в процесі виклику отримала копію значення змінної j.

- за адресою (передається вказівник на параметр, тобто у стек копіюється адреса);

```
...
int f(int* i)
{
int z ; (*i)=4;
z = *i;
i = new int[1000]; return z;

}
void main(void)
{
int j = 10; int* p = &j;
std::cout << "p = " << p << " j = " << j << '\n'; std::cout << "f = " << f(p) << '\n';
std::cout << "new j = " << j << " new p = " << p << '\n';
}
```

Передавання аргументів в тіло функцій за методом “адресою” передбачає копіювання в область стеку лише адреси змінною. За цією адресою функція може модифіковувати вихідне значення аргумента. Тому значення змінної j було зміненим.

Вказівники найширше використовуються при передачі аргументів функціям. Це дає змогу:

- зменшити витрати на копіювання змінної в стек функції;
- передавати у функцію великі за розміром програмні структури поза стеком даної функції, а, отже, уникнути ситуації переповнення його;
- підтримувати механізм двостороннього потоку даних між функцією, яка викликається, і тією, що викликає.

Оперують з вказівниками при передачі їх як параметрів функції, аналогічно як із звичайними параметрами.

```
...
```

```
const int sz = 10; int  array[sz]; int *ptr;
int get_size(int* );
void main(void)
{
for (int i = 0; i <= sz; i++)
array[i] = i; ptr = array;
std::cout << "Розмір масиву - " << get_size( ptr );
}
int get_size(int * pointer)
{
return sz * sizeof( pointer[0] );
}
```

- за посиланням. У цьому випадку у стек передається посилання (копіюється адреса), що дає змогу синтаксично використовувати даний параметр і як вказівник і як значення. Відповідно з'являється безпосередній доступ до вихідного значення змінної. Наприклад

```
...
int f(int i)
{
i = 4;
return i;
}
void main(void)
{
int j = 10;
std::cout << "j = " << j << '\n';
std::cout << "f = "<< f(j) << '\n';
std::cout << "new j = " << j << '\n';
}
```

За наведеним кодом змінна j не зберегла своє значення між викликами функції. Це зумовлене тим, що у стек копіювалась адреса, за якою у тілі функції був отриманий доступ до комірки пам'яті змінної j.

При передаванні за допомогою посилання, що це повинен бути існуючий об'єкт. Не можна за цим способом передавати значення із тимчасових змінних. Наприклад, неможна наведену у прикладі у функцію викликати із літеральною константою . Це означає, що виклик f(10) є неправильним. Літеральні константи передаються завжди через посилання на константу. Це означає, що для того щоб виклик був f(10) валідним функція повинна мати таку сигнатуру: int f(const int&).

Посилання найширше використовуються при передачі аргументів функціям. Це дає змогу:

- зменшити витрати на копіювання змінної в стек функції;
- передавати у функцію великі за розміром програмні структури поза стеком даної функції, а, отже, уникнути ситуації його переповнення;

Функція може приймати константні параметри, тобто правильною є конструкція типу: int f(const int b).

При передаванні параметрів через стек модифікатор const є зайвим оскільки працює сам стек (тобто задіються механізм копіювання значень). Модифікатор const використовується, як правило, при передаванні аргументів через вказівник, з метою заборони модифікації значення, на яке він посилається. Наприклад:

```
int f ( const int* ); void main(void)
{
const int * i = (const int*)5;
f(i);
}
void f(const int* m)
{
int k = 5; *m = k; // неправильне присвоєння
}
```

Подібно до АБП поряд із звичайними можна використовувати так звані анонімні параметри (АП), наприклад

```
int f(bool)
{
return 1;
}
void main(void)
{
f(true);
}
```

Викликати таку функцію треба обов'язково з з правильним набором параметрів. Проте звернутись до АП звичайними методами є неможливо, оскільки невідомим є ідентифікатор параметра.

Основним призначення АП є

- вирівнювання області стеку;
- забезпечення уніфікованих викликів функцій через вказівник;
- можливість задіяти розробником функції параметр в майбутньому. При цьому зникає необхідність зміни основного коду.

Іншою характеристикою функцій є повернення ними значень. Повернути із тіла функції можна лише одне значення і лише одного типу. Тип значення, яке повертає функція у C++11 завжди вказується явно.

Саме значення повертаються оператором повернення return. Аргумент цього оператора повинен строго відповідати типу значення, яке повертається.

Оператор повернення return, принаймні один, повинен завжди бути у тілі функція, яка повертає значення. Його виконання призводить до негайно завершення виконання коду тіла функції і організації процесу повернення у зовнішню функцію.

Якщо типом повернення є void, то функція не повертає значень. У цьому випадку оператор return у порожньому форматі є необов'язковий і може використовуватись лише для переривання виконання тіла функції, наприклад

```
void f()
{
// return 1; - помилка - не порожній формат
return;
}
```

Подібно до способів передавання значень у тіло функції існують способи повернення значень функціями.

У випадку повернення значень (не адресу) розрізняють об'єкти простого (скалярного) або складеного типу. Скалярні об'єкти характеризуються тим, що мають базовий тип і повертаються через регістри (зазвичай EAX). Складені об'єкти належать до структур або класів. При поверненні таких об'єктів, вони побайтово копіюються, наприклад

```
struct A
{
int i, j;
};
A a1 = {1, 2};
A a2 = {3, 4};
A f(); // прототип функції f
void main(void)
{
a1 = f(); // поля a2 побайтово скопіюються в a1
}
A f()
{
return a2; // функція повертає структуру типу a2
```

```
}
```

При поверненні з функції вказівника або посилання об'єкт, на який посилаються, повинен існувати після повернення з функції:

```
int& f1()
{
int i = 4;
return i;
}
int* f2()
{
int i = 3;
return &i;
}
```

Обидва випадки є невірними, оскільки область дії змінної і обмежена тілом функції f. Для вирішення цією проблеми треба змінну і оголосити глобальною або локальною статичною (static int i = 3), або повертати значення інших змінних.

11.4. Види функцій

11.4.1. Inline-функції

Декларація inline є модифікатором, який визначає вбудовуванні функції, тобто такі функції, виклики яких на етапі компіляції будуть замінені в точці виклику на код тіла функції.

За своїм змістом inline-функції є функціями звичайної структури з декількома (рекомендовано одним-двома) операторами і оператором return.

Компілятор на етапі компіляції замінює виклики inline-функції копією її тіла, а параметри функції – аргументами Це, вочевидь, не зменшить обсяг коду, проте зменшить витрати на організацію виклику функції, передавання аргументів та повернення результату, що, у свою, чергу пришвидшить виконання програми.

На Рис. 11.3 наведено приклади компіляції inlinei звичайної функцій разом з результатами роботи вбудованого відлагодника в режимі View CPU, у випадку використання компілятора BS5. З наведених сесій відлагодника, можна побачити, що у випадку використання модифікатора inline виклик функції був прибраний у компільованому коді.

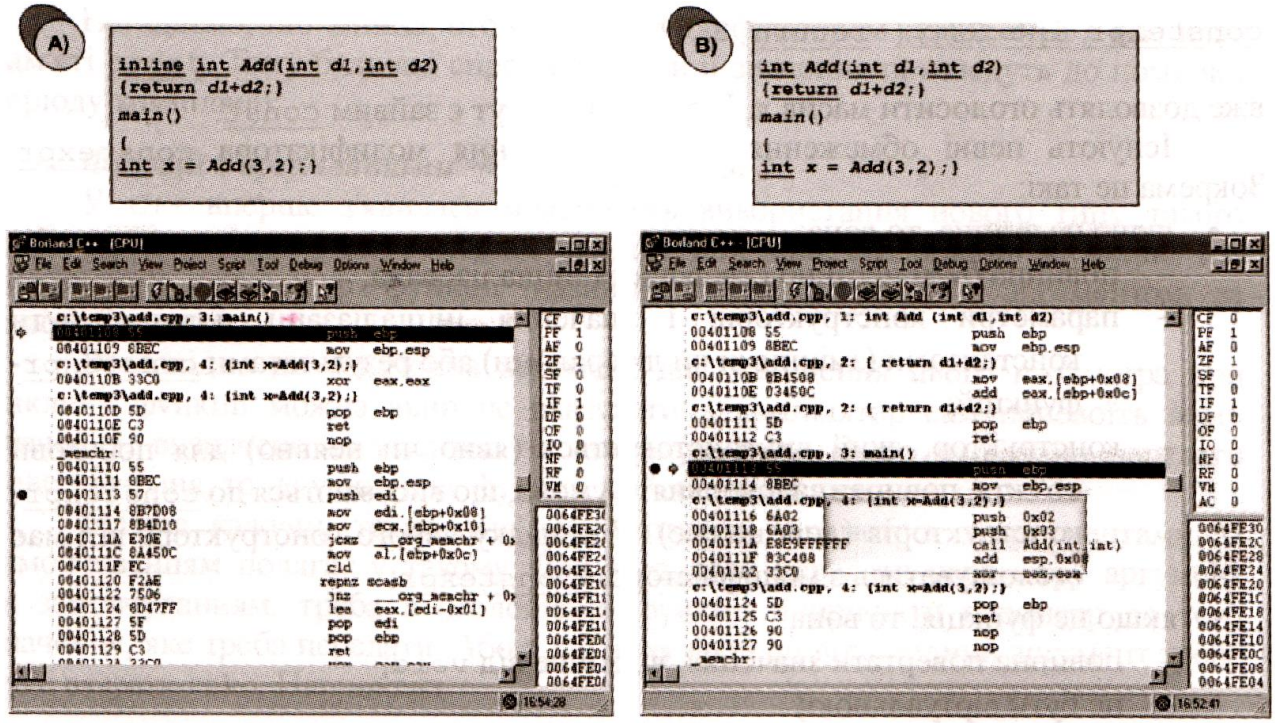


Рис. 11.3. Результат компіляції у випадку використання модифікатора inline

Наявність модифікатора inline не гарантує, що компілятор зробить дану функцію вбудовуванною. Модифікатор inline є лише запитом (декларацією бажання), а не командою. Для отримання позитивного результату дії модифікатора inline, функції треба будувати дуже короткими. Кількість і вид операторів, які можна використовувати у тілі, визначаються компіляторами.

Модифікатор inline дуже часто порівнюються з макропідстановкою define. Це не зовсім правильно, оскільки define є директивою рівня препроцесора, модифікатор inline інструкцією періоду компіляції. Використання модифікатора inline незалено від результату завжди гарантує перевірку правильності виклику. Тобто завжди буде здійснений контроль типів, якого ніколи немає у випадку макросу define. Проте макрос define має завжди гарантовану дію, у той час як модифікатор inline є лише декларацією побажань.

11.4.2. Узагальнені константні вирази

Ідея константних виразів полягає у трактовці і представленні виразів із константними змінними або літералами. Наприклад вираз 1+2 завжди повинен давати правильний результат.

Наявність константних виразів є одним із найбільш поширених способів оптимізація по швидкості виконання коду, який полягає у обчислення його результату лише один раз і виключно на етапі компіляції.

Стандарти мов C/C++ вимагає констані вирази при визначення деяких користувацьких типів, наприклад при визначенні статичних масивів. У цьому випадку розмірність масиву не може визначатись неконстатним виразом. Тому у коді

```
const int SZ(){ return 3; }
int m[ SZ() ];                          // помилкове оголошення
const int sz = 10;
int m1[ sz + 3 ];
```

правильним є визначення масиву m1 і неправильним масиву m. Результат роботи функції SZ() вважається невідомим на етапі компіляції (модифікатор inline та повертання константого цілого значення тут не допоможуть), оскільки по замовчуванні будь-яка функція може маніпулювати глобальними змінними, викликати інші функції і ін.

Для вирішення цієї колізії у C++11 введено зарезервоване слово (модифікатор функції) constexpr. Він розширює область застосування модифікатора const. Цей модифікатор вказує на те, що функція (чи об'єкт) повертає константне значення етапу компіляції. Наприклад, оголошення функції

```
constexpr int SZ(){ return 3; }
```

вже дозволять оголосити масив m. Модифікатор тут є зайвим const.

Існують певні обмеження на використання модифікатора constexpr. Зокрема це такі:

- якщо це змінна, то вона:
  - повинна бути негайно створена та ініціалізована;
  - параметри конструкторів і значення ініціалізації повинні бути константами (змінними чи літералами) або результатами constexpr-функцій;
  - конструктор, який використовується (явно чи неявно) для побудови об'єкта, повинен задовольняти умови, що висуваються до constexpr-конструкторів. У випадку явного конструктора він повинне оголошуватись з модифікатором constexpr.
- якщо це функція, то вона:
  - повинна повертати значення відмінне від void;
  - не бути віртуальною;
  - не повинна викликатись до моменту свого визначення в модулі компіляції;
  - мати параметри простих типів;
  - мати тіло, яке може складатись із таких конструкцій:
  - null вирази (порожній вираз, тобто ;)
  - декларації static_assert;
  - декларацій typedef або аліасів (псевдонімів), які не визначають клас або перелік;
  - декларацій using та директив using;
  - мати виключно один оператор return;

– повертати лише константу (змінну чи літерал) або constexpr-об'єкт чи результат constexpr-функції.

• якщо це конструктор, то:

– він повинен мати параметри лише простого типу;

– клас не повинен мати віртуальних батьківських класів;

– повинен бути визначеним до моменту використання у модулі компіляції;

– тіло відповідати вимогам, які висуваються до тіла constexpr-функцій за винятком вимог до оператора return.

– тіло не повинно містити (явно чи неявно) блоків try ділянок програмного коду з можливостями перехоплення виняткових ситуацій;

– кожен батьківський клас та нестатичний член класу повинні ініціалізовуватись ініціалізатором складених типів (явна ініціалізація) або ініціалізатором списків initializer_list.123 Більше цього кожен конструктор, який викликається у даному, повинен також бути constexpr-конструктором;

– операція неявного приведення повинна повертати результатом константний вираз;

Наостанок нагадаємо, що змінна оголошена з модифікатором constexpr має клас пам'яті const. Тому будь-які спроби змінити її значення призведуть до помилки періоду компіляції.

11.4.3.Функції з аргументами по замовчуванні

У C++ вперше з'явилась можливість використання нового типу параметра такого, значення якому можна вказати вже при оголошенні функції. Так визначений параметр функції називається параметром з аргументом по замовчуванні.

Відповідно до означення аргумент для заміщення цього параметра при виклику функції можна явно не вказувати – компілятор сам присвоїть йому значення, яке визначене аргументом по замовчуванні. Якщо необхідно передати нове значення, то це у виклику функції здійснюється явно.

Іншими словами, технологія використання параметрів з аргументами по замовчуванні полягає у наступному. Для того, щоб передати параметру аргумент по замовчуванні, треба в оголошені функції прирівняти параметр до того значення, яке треба передати. Або у виклику функції відповідний аргумент треба буде вказати явно. Наприклад

Приклад № 11.5.

```
#include <iostream>
double square(int a, int b = 0)
{
return (b) ? a * b : a * a;
}
```

```
void main(void)
{
std::cout << "Площа квадрата: " << square(1) << '\n';
std::cout << "Площа прямокутника: " << square(1, 2) << '\n';
}
```

У наведеному коді про використання аргументів по замовчуванню вказано у визначенні функції. Вже при виклику функції можна опускати заміщення параметрів, які приймають аргументи по замовчуванні (зокрема для параметра b).

У випадку оголошення прототипу аргументи по замовчуванні оголошуються лише в прототипі. Зокрема код з Прикл. № 11.5 видозміниться так

Приклад № 11.6.

```
#include <iostream>
double square(int a, int b = 0);
void main(void)
{
std::cout << "Площа квадрата: " << square( 1 ) << '\n';
std::cout << "Площа прямокутника: " << square( 1, 2 ) << '\n';
}
double square(int a, int b)
{
return (b) ? a * b : a * a;
}
```

Прототип функції square можна було записати ще коротше, зокрема так

```
double square(int, int = 0);
```

Використовуючи аргументи по замовчуванні треба дотримуватись двох правил:

- при оголошенні функції усі її аргументи можуть приймати аргументи по замовчуванні;
- коли параметру присвоюється аргумент по замовчуванні, то й усім наступним параметрам треба призначати параметри по замовчуванні; усі звичайні параметри оголошуються попереду параметрів з аргументами по замовчуванні;
- коли використовується значення для явного заміщення параметра з аргументом по замовчуванні, то усі попередні параметри з аргументами по замовчуванні також повинні бути явно заміщеними аргументами. Іншими словами, якщо значення аргумента по замовчуванні задане явно, то заданими явно повинні бути усі попередні аргументи незалежно від типу їх оголошення.

Наприклад, якщо функція має такий прототип void F(int a, int n = 10, int m = 11, int z = 12), то допустимими викликами є такі: F(1); F(1, 2); F(1, 2, 3); F(1, 2, 3, 4); . Відповідно недопустимим викликами є такий F(1, , 3); і йому подібні.

Наостанок зазначимо, що існування аргумента по замовчанню не пришвидшує виклик функції. Функція буде викликатись за звичайною процедуро і заміщуватись будуть усі параметри. Параметр, який використовує значення по замовчуванні також. Відповідний йому аргумент буде передаватись у стек неявно.

11.4.4.Функції із змінною кількістю параметрів

Поряд із звичайними існують функції із змінною кількістю параметрів. При оголошенні таких функцій використовується спеціальне позначення: '...', наприклад

```
int f( int count, ... );
```

Для доступу до параметрів такої функції використовують макроси, оголошені у ФЗ stdarg.h: va_list, va_start, va_arg, va_copy, va_end.

Окрім цього для коректного використання макроса va_start йому треба передати останній явно заданий параметр для визначення початку списку змінних параметрів.

Порядок використання описаних вище макросів є строгим, що ілюструє наступний приклад:

```
Приклад № 11.7.
#include <iostream>
#include <stdarg >
void f(int, ... );
void main(void)
{
f(1, 1);
f(2, 1, 2);
f(3, 1, 2, 3);
}
void f(int i, ... )
{
// організація вказівника на список параметрів
va_list param;
// вказівник напочаток
va_start(param, i);
for (int j = 1; j <= i; ++j)
std::cout << "argument #" << j << " value " <<
va_arg(param, int) << '\n';
```

```
        // звільнення вказівника
va_end(param);
std::cout << "----------------\n";
}
```

Макрокоманда va_arg використовується у цьому прикладі для перегляду списку і вибірки з нього параметрів функції. При цьому вказівник на список параметрів пересувається розмір типу, який задається другим параметром функції va_arg. Оскільки жодних перевірок на те чи досягнуто кінець списку параметрів вона не здійснює, то для коректного зчитування чергового параметра їй необхідно явно передати його тип. Завданням типу є визначення розміру ділянку пам'яті, на яку вказує param, та формату повернення.

Макрокоманда void va_copy(va_list dest, va_list src) використовується для копіювання списку параметрів, який адресується параметром src у src, наприклад

```
...
va_list p1;
va_start(p1, count);
va_list p2;
va_copy(p2, p1);
...
```

Перед поверненням із функції повинен бути викликаний макрос va_end. Це зумовлено тим, що va_arg може змінити стек так, що успішне повернення стане неможливим. Макрос va_end нівелює зміни зроблені макросом va_arg.

Використання макросів stdarg.h можна уникнути і аргументи функції вичитувати засобом звичайного вказівника. Для початкової ініціалізація такого вказівника необхідно адресу останнього розміщеного в стеку параметра збільшити на розмір його типу, наприклад

```
void f(int i, ... )
{
int* p = &i +1; // організація вказівника
for (int j = 1; j <= i; p++, ++j)
std::cout << "argument #" << j << "     value " << *p << '\n';
std::cout << "----------------\n";
}
```

У ранніх версіях мови C++ для функцій із змінною кількістю параметрів обов'язковим було явне оголошення принаймні одного параметра (може бути довільна кількість) у звиклий спосіб. Вважалось, що функція "не знала" кількості параметрів при конкретному виклику, а тому призначенням явно заданих

параметрів забезпечувало реалізацію механізму визначення реальної кількості параметрів при кожному конкретному виклику.

Для використання таких функцій на практиці, треба враховувати структуру стекової пам'яті, яка реалізовується компілятором. Наприклад для компілятора MSVS2012 можна написати такий код

Приклад № 11.8.

```
#include <iostream>
typedef unsigned int UL;
void f( ... )
{
UL fEBP;
__asm
{
mov fEBP, EBP
}
int* p = (int*)fEBP + 2;
for (int j = 1; j <= 4; p++, ++j)
std::cout << "argument #" << j << " value " << *p << '\n';
std::cout << "----------------\n";
}
void main(void)
{
f(1, 2, 3, 4);
        }
```

```
#include <iostream>
typedef unsigned int UL;
void f( ... )
{
```

Описаний підхід (із використанням регістра глибини стеку) використовувався у функціях із порожнім списком параметром (без використання зарезервованого слова void) у мові C.

11.4.5. Лямбда-функції

Починаючи із C++0x на рівні стандарту мови реалізована концепція лямбда- функцій (лямбда-виразів). Суть цієї концепції полягає у забезпеченні механізму створення в поточному коді неіменованих функторів (функціональних об'єктів). Іншими словами – технологія лямбда виразів дозволяє об'єднати в одному операторі створення фунціонального об'єкта і зв'язування його з кодом зворотного виклику під час створення.

Типова структура лямбда-функції є такою

```
[маска] (параметри) mutable throw() -> тип повернення
{тіло функції}(аргументи);
```

Наприклад

```
int x = 1, y = 2;
auto i = [=] { return x + y; };
int j = [] (int k) -> int { return k + 1; } (y);
```

Маска визначає спосіб доступу до зовнішніх змінних у тілі функції. Доступними є будь-які змінні. Ті, які використовуються у тілі функції, складають поняття замикання лямбда-функції. Для них підтримуються лише два способи доступу. Перший з них за значенням, а другий за посиланням. Наприклад [&x, y]. Тут змінна x буде доступна за посиланням, а змінна y – за значенням. Порожня маска означає, що доступу до змінних не має, а маска [=] визначає доступ за значенням. Наприклад та сама маска може бути записана у виглядах: [&, y], [=, &x]. Наведемо перелік допустимих масок:

- [] – доступу до зовнішніх змінних немає;
- [=] – доступ до усіх зовнішніх змінних є за значенням;
- [&] – доступ до усіх зовнішніх змінних є за посиланням;
- [x, y] – доступ до x і y є за значенням;
- [&x, &y] – доступ до x і y є за посиланням;
- [x, &y] – доступ до x за значенням, а до y – за посиланням;
- [=, &x, &y] – доступ до усіх зовнішніх змінних, окрім x та y є за значенням; доступ до x і y є за посиланням;
- [&, x, &y] – доступ до усіх зовнішніх змінних, окрім x, є за посиланням.

Якщо лямбда-функція використовується у класах, то за допомогою маски у тіло функції можна передавати вказівник на об'єкт, наприклад [this].

Треба пам'ятати, що зовнішні змінні, які доступні в лямбда-функції, можуть змінювати свої значення тільки за умови доступу через посилання. Наприклад

```
auto i = [&x, y]
{
x++;
// y++; помилка y доступна за значенням
return x + y;
};
```

Список параметрів є типовим для звичайних функцій за винятком:

- параметри не можуть приймати значення по замовчуванні,
- параметри не можуть бути анонімними
- кількість параметрів є обмеженою.
- параметром може бути інша лямбда-функція Наведемо приклад

```
auto i = [&x, y] (int k)
{
x = 10;
k = 20;
return x + y + k;
}(x);
```

Порожній список параметрів визначається відсутністю списку параметрів, або порожнім списком, наприклад

```
auto i = [=] () { return x + y; };
```

Якщо список параметрів є непорожнім, то список аргументів повинен бути також не порожнім. Кількість аргументів строго визначається кількістю параметрів. Підтримується усі три способи передавання значень в тіло функції, наприклад

```
int i = [=] (int k, int& l, int* z) -> int
{
return (++k) + (++l) + (++(*z)) ;
} (x, x, &x);
```

Модифікатор mutable визначає доступність для зміни усіх глобальних змінних, незалежно від маски, наприклад

```
auto i = [=] () mutable
{
x = 10;
y = 20;
return x + y;
};
```

Модифікатора mutable може бути опущений і його доступність самого модифікатора mutable визначає явно заданням списку параметрів (порожнім чи непорожнім).

Оператор throw визначає специфікацію виключення. Явне задання модифікатора може бути опущене. Тоді специфікатором буде використовується

noexcept. Таким також буде специфікатор у випадку явного вказання порожнього оператора throw(). Цим визначається заборона генерації виключення.

Доступність оператора throw визначається подібно до модифікатора mutable.

Тип повернення є аналогічний типу повернення функції і визначає значення, яке буде повертатись лямбда-функцією. На відміну від звичайних функцій:

- тип повернення може не вказуватись (тоді тип – void),
- якщо тип повернення вказується, то список параметрів повинен бути не порожнім.

За стандартом мови лямбда-функція будує тимчасовий анонімний об'єкт типу ClosureType. Проте на практиці тип самої лямбда-функції може залежить реалізації компілятора. За стандартом мови лямбда-функція будує тимчасовий анонімний об'єкт типу

Якщо виникає потреба передати лямбда-функцію параметром, то вона повинна бути шаблонного типу або збережена з використанням шаблону std::function. Зарезервоване слово auto дозволяє локально зберегти лямбда-функцію. Наприклад

```
auto i = [=] { return x + y; };
std::cout << typeid(i).name();
```

Лямбда-функція може повертати значення лише одного типу. Подібно до звичайних функцій воно може бути проігнороване, наприклад

```
[=] { return x + y; };
```

Тіло функції є подібним до тіла звичайної функції. Допускається створення локальних змінних, наприклад

```
[=]
{
int k = 5;
if ( k > 5 ) return x + y;
else
return x - y;
};
```

Тіло функції може бути порожнім. Оператор повернення return є не обов'язковим, якщо тип повернення не вказаний явно, наприклад

```
auto i = [=] () { };
```

Наведемо приклад сумісного використання функцій з аргументами по замовчуванні та лямбда функцій

```
...
void f01( const std::string& arg = "world")
{
std::cout << "Hello, " << arg << '\n';
}
auto f02() -> void(*)(const std::string&)
{
return f01;
}
void (*f03())(const std::string&)
{
return f01;
}
```

У наведеному коді функція f01 є функція з аргументами по замовчуванні, яка нічого не повертає. А лямбда функція f02 повертає вказівник на функцію f01. У попередніх версіях мови C++ таке завдання вирішувалось за допомогою функції f03.

З врахуванням явного задання складових визначення лямбда-функцій, можна виділити їх 4 типи визначення:

– [маска] (параметри) mutable throw() -> тип повернення {тіло функції}(аргументи); - повне визначення лямбда-функції;

– [маска] (параметри) -> тип повернення {тіло функції}(аргументи); - визначення const лямбда-функції;

– [маска] (параметри) -> {тіло функції} (аргументи); - визначення лямбда-функції із автоматичним визначенням типу. Якщо оператор return заданий явно, то тип визначається за його виразом і decltype. У противному випадку тип повернення void.

– [маска] {тіло функції}; - визначення лямбда-функції без параметрів;

11.4.6. Перевантажені функції

У C++ введена можливість використовувати одне ім'я функції для оголошення і побудови різних версій функцій. Така можливість називається перевантаженням функції. Це означає, що на один і той же ідентифікатор навантажено декілька функцій, які відрізняються типом і кількістю аргументів. У ранніх версіях компіляторів C++ для створення перевантажених функцій вимагалось використання зарезервованого слова overload. Сучасні версії його використання не підтримують.

Для того, щоб коректно використовувати механізм перевантаження функції необхідно дотримуватись правил:

- кожна версія функція повинна мати відмінну від інших коротку сигнатуру, тобто кількість, послідовність і тип параметрів функції.
- якщо функція містить параметри з аргументами по замовчуванні, то компілятор не вважає їх частиною сигнатури. Тому може виникнути ситуації неоднозначного виклику і компілятор видасть повідомлення про помилку. Для перевантаження можна використовувати довільне ім'я функцій в межах ОД. У випадку, коли компілятор зустрічає різні звертання до перевантаженого імені він шукає найбільш прийнятну для використання функцію. Результат цього пошуку відображається у відкомпільованому коді. Тому при перевантаження зростають витрати на компіляцію. Проте, під час виконання програми звертання до перевантажених функцій здійснюється як до звичайних і додаткових часових витрат не виникає.

Наведемо приклад перевантаження функцій

Приклад № 11.9.

```
#include <math>
#include <iostream>
double area(double);
unsigned int area(unsigned int);
double area(double,double);
double area(double,double,double);
void main(void)
{
std::cout << "Area of a circle" << area( (double)1 ) <<
"\nArea of a square: " << area( (unsigned int)2 ) <<
"\nArea of a rectangle: " << area(3, 4) <<
"\nArea of a triangle: " << area(5, 6, 7) ;
}
        double area(double d1)
{
return 3.14 * d1 * d1;
}
unsigned int area(unsigned int d1)
{
return d1 * d1;
}
double area(double d1,double d2)
{
return d1 * d2;
}
```

```
double area(double d1,double d2,double d3)
{
double p = (d1 + d2 + d3)/2;
return sqrt(p*(p - d1)*(p - d2)*(p - d3));
}
```

У наведеному прикладі уніфіковано процедуру обчислення площі геометричної фігури. Фактично розробнику достатньо пам'ятати лише ім'я цієї процедури. Тип геометричної фігури буде вибраний автоматично за кількістю аргументів у точці виклику.

У Прикл. № 11.9 треба звернути увагу на два важливих моменти:

- для вирішення проблеми однозначного вибору версії функції у викликах використовується явне приведення типів. Використання неявного приведення може стати причиною логічної помилки алгоритму виконання програми;
- у випадку відсутності прототипу unsigned int area(unsigned int); навіть при наявності її визначення виклики area((double)1) і area((unsigned int)2) призведуть до виклику версії функції double area(double).

Без жодних окремих обмежень допускається перевантаження inline-функцій.

Наостанок виділимо основні проблеми, які можуть виникати в процесі перевантаження функцій. Усі переваги технології перевантаження можуть бути знівельовані такими ситуаціями:

- перевантажена функція може викликатись з неоголошеними при перевантаженні аргументами;
- тип аргумента у виклику перевантаженої функції можна трактувати неоднозначно. В такому випадку треба явно здійснювати приведення типів;
- намаганням перевантажити функції, єдиною відмінністю яких є використання в одному випадку параметра-посилання, а в іншому параметра-значення. Тут треба зважати на те, що не існує синтаксичних відмінностей між викликом функції за значенням і викликом функції за посиланням. Наведені ситуації не висувають непереборних перепон в процесі програмування, а є свідченням того, що до використання механізму перевантаження функцій треба підходити дуже виважено.

11.4.7. Callback функції

Callback функцією або функцією із зворотнім викликом називається функція, яка у якості параметрів приймає іншу функцію з метою реалізації концепції 'передавання виконавчого коду аргументом іншого виконавчого коду'. Callback виклик дозволяє всередині одних процедур виконувати код, який визначений у форматі інших процедур і є недоступний для прямого виклику. Наприклад

Приклад № 11.10.

```
#include <iostream>
typedef unsigned short US;
const US sz = 7;
int callback_fun(int *m, US len, int(*comparator)(int, int) )
{
int r = m[0];
for(US i = 1; i < len; ++i)
r = comparator(r, m[i]);
return r;
}
int max(int i, int j) { return i < j ? j : i; }
int min(int i, int j) { return i > j ? j : i; }
void main(void)
{
int data[sz] = {3, 2, 1, 0, -1, -2, -3};
std::cout << "Max element is " <<
callback_fun(data, sz, max) << '\n';
std::cout << "Min element is " <<
callback_fun(data, sz, min) << '\n';
return 0;
}
```

У наведеному коді функція callback_fun є функцією із зворотнім зв'язком. Її повний функціонал забезпечується третім параметром, який, у свою чергу, є вказівником на функцію. Це означає, що через заміщення цього параметра у викликах callback_fun функції буде передаватись програмний код, який сформований у форматі функції. Цим самим вдалось забезпечити реалізацію однією функцією двох дій (пошуку максимального і мінімального елемента), при мінімальних витратах на базисну операцію (у прикладі це операції порівняння на менше і більше).

Описаний підхід дає можливість створювати за допомогою зворотного виклику функцій загального призначення, які призначені для уникнення створення наборів функцій однакової структури, але з різними підзадачами. Такі загальні функції іноді ще називають функціями загального призначення.

Ще одним призначенням callback функцій є реалізація ідеї повторного використовуваного коду. Один раз написана і перевірена функція загального призначення забезпечить безпомилкове багатократне виконання базового коду.

Використання callback функцій дозволяє структурувати програмний код. Здійснюється це за рахунок того, що поведінка програми із незмінним кодом (тіло callback функції) може змінюватись (іноді навіть дуже сильно) завдяки функціоналу, що передається у цей код. На практиці це здійснюється створенням

нового функціоналу, або додавання до ланцюжка викликів ще однієї функції, яка цей функціонал реалізовує.

Лекція 12. Послідовність виклику функцій. Вказівники на функції. Масиви вказівників.

12.1. Порядок виклику функції

Порядок виклику (calling conventions) визначає процедури виклику та завершення виконання функції, зокрема:

- передавання даних у тіло функції способом через стек, регістри або сумісним використанням і стеку і регістрів;
- порядок передавання аргументів. При використанні стеку порядок занесення даних у стек. При використанні регустрів – порядок співставлення параметрів і регістрів. Розглядають два способи. Перший з них – це прямий порядок. За ним аргументи розміщуються у тому порядку, у якому вони оголошувались у визначенні функції. Другий спосіб – це зворотній порядок, тобто протилежний до оголошення у списку параметрів. За ним у вершині стеку завжди буде розміщуватись перший параметр функції.
- порядок повернення вказівника стеку у вихідну позицію (очистка стеку). Тут можливі також два варіанти. За першим стек у вихідну позицію переводить сама функція. Така схема є дуже зручно, оскільки команди очистки стеку просто додаються в кінець тіла функції. За другим варіант стек чистить зовнішня функція (тобто така, яка викликала дану). Перевагою другого підходу є простота реалізації змінної кількості параметрів функції.
- команди виклику функції (call far, call near, pushf/call far) та повернення у зовнішню функцію (команди retn, retf, iret).
- набір регістрів, які значення яких підлягають відновленню після виконання тіла функції і перед поверненням.

У цьому контексті існує декілька модифікаторів, які визначаються цей порядок. Усі вони не є визначені стандартом (за винятком inline), але набули великого поширення у різноманітних компіляторах. Тому варто розглянути основні з них.

Серед розглянутих далі модифікаторів для мови C++ основним (по замовчуванні) є модифікатор cdecl. Його принциповою особливістю є те, що на відміну від інших, cdecl-функція не зобов'язана перед поверненням відновлювати значення сегментних регістрів та регістрів ESP і EBP.

Модифікатор cdecl.

1. Декларація cdecl є основним модифікатором мови C і визначає виклик функцій за послідовністю мови C, при якій зберігається чутливість до регістрів імен функцій.

Виклик функції за послідовністю мови C визначає зворотний стосовно оголошення порядок занесення параметрів у стек (справа наліво), а також покладає відповідальність за його очищення на програму, яка викликає дану функцію.

Як приклад розглянемо програму.

Приклад № 12.1.

```
int Funct(int i1, int i2, int i3)
{
return i1 + i2 + i3;
}
void main(void)
{
int i = Funct(1, 2, 3);
}
```

Результат її компіляції компілятором BS5 наведено на Прикл. № 12.1. Вбудованим відлагодником в режимі View CPU отримано відповідний набір асемблерних інструкцій наведених на Прикл. № 12.1, які ілюструють порядок виклику cdecl-функції Funct().

Використання модифікатора cdecl дає змогу будувати функції із змінною кількістю параметрів.

Модифікатор pascal.

Декларація pascal є модифікатором, який визначає послідовність виклику функцій стандартом мови Pascal. При цьому усі символи в іменах pascal-функцій приводяться до машинного регістра.

Виклик функції з модифікатором pascal визначає прямий порядок занесення параметрів функції у стек, тобто зліва направо. Відповідальність за очищення стеку покладена на саму функція. Змінні, які повинні змінювати своє значення, повинні передаватись виключно за посиланням. Наприклад

Приклад № 12.2.

```
int pascal Funct(int ,int , int );
void main(void)
{
int i = Funct(1, 2, 3);
}
int pascal Funct(int i1, int i2, int i3)
{
return i1 + i2 + i3;
}
```

Особливістю pascal-функцій є те, що у ній неявно створюється перший параметр, через який повертається значення.

Виклик функцій за pascal послідовністю був основним у ранніх версіях Windows. У нинішніх реалізація цієї ОС основним є спосіб виклику, який визначається модифікаторами stdcall/winapi.

Модифікатори stdcall/winapi**.**

Ці модифікатори є близьким до cdecl. Аргументи передаються через стек у зворотному порядку, тобто зправа наліво. Очищенням стеку займається зовнішня функція.

Модифікатори stdcall/winapi використовується при виклику функцій WinAPI. Тому їх застосування стосовно функцій із системний викликом є обов'язковим.

Модифікатор fastcall/ msfastcall**.**

Цей модифікатор називають модифікатором швидкого виклику. На відміну від усіх решти модифікаторів, які мають визначення у різних специфікаціях, цей модифікатор жодної специфікацією не описується. Особливістю його роботи є те, що, якщо це можливо за розміром, аргументи у тіло функції і усі проміжні результати виконання тіла функції розміщуються у регістрах. Це означає, що стек може бути взагалі не потрібним. Параметри fastcall-функції мають по замовчуванні клас пам'яті register. Очищення стеку повинна здійснювати зовнішня функція.

Порядок занесення аргументів при виклику fastcall-функції є різним для компіляторів різних виробників. Так у компіляторах Borland (модифікатор fastcall) аргументи передаються завжди у порядку зліва направо. Якщо аргументів менше трьох, то використовуються регістри (EAX, EDX, ECX), а якщо більше – то завжди через стек.

У компіляторах *Microsoft* та *GCC* (використовуються обидва слова) порядок занесення аргументів визначається їх кількістю. Якщо аргументів не більше двох, то порядок є прямим з використанням регістрів (EDX, ECX). Якщо більше, то порядок є зворотнім і використовується завжди стек.

Нагадаємо, що в усіх описаних випадках використання регістрів, передбачається, що змінна має такий розмір, що дозволяє її розміщеня в регістрах.

Модифікатор thiscall**.**

Модифікатор використовується у випадку використання засобів ООП. Він стосується організації викликів методів об'єктів. За способом організації виклику цей модифікатор є аналогічний модифікатору stdcall. Але його особливістю є спосіб передавання вказівника this. При використання модифікатора thiscall вказівник this буде передаватись через регістр ECX.

Модифікатор interrupt**.**

Декларація interrupt є модифікатором, який визначає виклик функції безпосередньою апаратурою через механізм переривань. Звичайні функції не можуть оголошуватись як interrupt-функції. Більше того функції, які викликаються interrupt функціями не можуть оголошуватись з цим модифікатором.

Interrupt-функції оголошуються без аргументів і не повертають значень, оскільки механізм виклику є апаратним і не обробляє повернення з функції.

Значення у ці функції передаються і повертаються з них виключно через регістри.

Перед свої використанням interrupt-функція повинна бути встановлена через бібліотечну функцію setvect() на відповідний вектор. Як обробник переривання вона напочатку роботи повинна зберігати значення регістрів, в кінці їх відновлювати та містити інструкцію ret.

Використання визначення порядку виклику є необхідним у випадках

- оптимізації програми,
- викликів функцій системних бібліотек
- спряженні програми із іншими програмними модулями, які розроблялись на іншій мові чи іншими інструментальними засобами.

12.2. Вказівники на функції. Масиви вказівників

Як вже зазначалось вище функція є не що інше як вказівник в області коду. Тому природнім є розгляд питання організації сторонніх вказівників на функції.

Мова C++ підтримує механізм організації таких вказівників. Їхніми значеннями є адреси функцій, тобто адреси перших виконавчих операторів.

Загальний синтаксис оголошення вказівника на функцію має вигляд

```
тип_поверн (*pointer)(спис_параметрів)[ = значення];
```

Тоді загальний синтаксис виклику функції через вказівник виглядає так

```
pointer(спис_аргум);      // новий формат
( *pointer)(спис_аргум); // старий формат (підтримується)
```

Складові тип_поверн і спис_параметрів є типовими для визначення сигнатур функцій у мові C++. Не підтримується лише сигнатура функцій із аргументами по замовчуванні.

Треба пам'ятати, що усі модифікатори (за винятком cdecl) у визначеннях функцій, які будуть викликатись через вказівник, повинні бути вказані у його визначенні.

При ініціалізації вказівників на функції іменем конкретної функції треба забезпечити відповідність типу, який повертає функція, і списку її параметрів до оголошення вказівника. Ініціалізація може здійснювати при створенні вказівника, або у довільному місці програми за форматами

```
pointer = f;  // новий формат
pointer = &f;         // старий формат (підтримується)
```

де f – функція, сигнатура якої збігається із сигнатурою, яка, у свою чергу, була вказана у визначенні вказівника.

Після ініціалізації вказівника на функцію його можна використовувати для виклику цих функцій, наприклад

Приклад № 12.3.

```
int f1(int a, double b) // визначення функції
{
return (a + int(b));
}
int f2(int a, double b)
{
return (a - int(b));
}
void main(void)
{
int (*p)(int, double);        // оголошення вказівника на функцію
p = &f1;       // організація вказівника на f1
int i = p(1, 2);         // виклик f1 через вказівник
p = f2;          // переадресація вказівника на f2
i = p(1, 2);    // виклик f2 через вказівник
i = (*p)(1, 2);          // виклик в старому форматі
}
```

Сигнатура функції у визначенні вказівника визначає його тип. Наприклад, оператор std::cout << typeid(p).name(); для вказівника p із Прикл. № 12.3, виведе на екран таку інформацію про тип: int (cdecl*) (int, double). Саме цей запис і є іменем типу змінної p. У визначенні типу присутній модифікатор cdecl був заданий неявно. Вказівник f1, на відміну від p, є константним вказівником.

Окремо варто розглянути таку цікаву конструкцію як масив вказівників на функції. Ця конструкція реалізовує ідею (концепцію) керованого таблицями коду: замість умовних команд чи конструкцій вибору виконувана функція вибирається в залежності від змінної (чи комбінації змінних) стану. Використання цієї концепції є особливо зручним при частих додаваннях чи видаленнях функцій з таблиці, а також при динамічному створенні чи зміні самої таблиці.

Для того, щоб викликати потрібну функцію, визначену у масиві, треба

- проіндексувати масив,
- розіменувати отриманий вказівник.

Для прикладу організації та використання масиву вказівників, приведемо такий код:

Приклад № 12.4.

```
#include <iostream>
#define MF(N) void N(){\
```

```
std::cout << "Виклик функції "#N ; }
MF(a); MF(b); MF(c); MF(d);
void (*mf[])() = { a, b, c, d };       // створення масиву
void main(void)
{
while (1)
{
std::cout << "Введіть значення від a до d або q для виходу ";
char sw; std::cin >> sw;
if ( sw == 'q' ) break;
if ( ( sw >= 'a' ) && ( sw <= 'd' ) )
( *mf[sw - 'a'])();    // виклик функції
}
}
```

У наведеному прикладі за допомогою препроцесорного макроса #define створюється декілька фіктивних функцій. Це робиться з метою уникнення потреби створення набору функцій. Далі створюється масив вказівників на функції з автоматичною ініціалізацією по списку. Як неважко переконатись, додавання чи знищення функції з таблиці (що є зміною функціональних можливостей програми, досягається мінімальним об'ємом програмного коду.

Перевірити тип вказівника на функцію можна за допомогою вже описаного шаблону std::is_pointer

```
#include <type_traits>
#include <iostream>
void f (int) {}
...
std::cout << std::boolalpha
<< std::is_pointer<void (*) (int)>::value << '\n'
<< std::is_pointer<decltype(f)>::value << '\n'
```

Інший спосіб визначення типу полягає у використання шаблону std::is_ function.

Лекція 13. Стрінги мови С.

С ++ не містить стандартного типу даних «стрінг». Замість цього він підтримує масиви символів, що завершуються нуль-символом. Бібліотека містить функції для роботи з такими масивами, успадковані від C і описані в заголовному файлі <string.h> (<cstring>). Вони дозволяють досягти високої ефективності, але дуже незручні і небезпечні у використанні, оскільки вихід за межі стрінгу не перевіряється.

Тип даних string стандартної бібліотеки позбавлений цих недоліків, але може програвати масивам символів в ефективності. Основні дії зі стрінгами виконуються в ньому за допомогою операцій і методів, а довжина стрінга змінюється динамічно відповідно до потреб. Для використання класу необхідно підключити до програми заголовки <string>. Розглянемо приклад:

```
#include <cstring>
#include <string>
#include <iostream>
using namespace std;
Int main () {
        char c1 [80], c2 [80], c3 [80]; // Стрінги з завершальним нулем string
    sl, s2. s3;
        // Присвоєння стрінгів
        strcpy (cl, "old string one");
        strcpy (c2, c1):
    sl = "new string one";
    s2 = s1;
    // Конкатенація стрінгів
    strcpy (c3, c1):
    strcpy (c3. c2);
    s3 = s1 + s2:
    // Порівняння стрінгів
    if (strcmp (c2, C3) <0) cout << c2;
    else cout << c3:
    if (s2 <s3) cout << s2;
    else cout << s3;
}
```

Як видно з прикладу, виконання будь-яких дій зі стрінгами старого стилю вимагає використання функцій і менш наочно. Крім того, необхідно перевіряти, чи достатньо місця в стрінгу-приймачі при копіюванні, тобто фактично код роботи зі стрінгами старого стилю повинен бути ще довшим.

Стрінги типу string захищені від виходу інформації за їх межі, і з ними можна працювати так само, як з будь-яким вбудованим типом даних, тобто за допомогою операцій. Розглянемо основні особливості і прийоми роботи зі стрінгами.

13.1. Конструктори і присвоювання стрінгів

У класі string визначено кілька конструкторів. Нижче в спрощеному вигляді наведено заголовки найбільш уживаних:

```
string ();
string (const char *);
string (const char * .int n);
string (string &);
```

Перший конструктор створює порожній об'єкт типу string. Другий створює об'єкт типу string на основі стрінгу старого стилю, третій створює об'єкт типу string і записує туди n символів зі стрінгу, що зазначено першим параметром. Останній конструктор є конструктором копіювання, який створює новий об'єкт як копію об'єкта, переданого йому в якості параметра.

В класі string визначені три операції присвоювання:

```
string & operator = (const string & str);
string & operator = (const char * s);
string & operator = (char c);
```

Як видно із заголовків, стрінгу можна присвоювати інший стрінг типу string, стрінг старого стилю або окремий символ, наприклад:

```
string s1;
string s2 ( "Вася");
string s3 (s2);
s1 = 'X';
s1 = "Вася";
s2 = s3;
```

13.2. Операції

Нижче наведені допустимі для об'єктів класу string операції:

Операція	Дія	Операція	Дія
=	присвоювання	>	більше
+	конкатенація	> =	більше або дорівнює
==	рівність	[]	індексація

! =	нерівність	<<	виведення
<	менше	>>	введення
<=	менше або дорівнює	+ =	додавання

Синтаксис операцій і їх дія очевидні. Розміри стрінгів встановлюються автоматично так, щоб об'єкт міг містити присвоєне йому значення. Треба відзначити, що для стрінгів типу string не дотримується відповідність між адресою першого елементу стрінга і ім'ям, як це було у випадку стрінгів старого стилю, тобто &s[0] не дорівнює s.

Крім операції індексації, для доступу до елементів стрінгу визначена функція at:

```
string s ( "Вася");
cout «s.at (l);        // Буде виведений символ а
```

Якщо індекс перевищує довжину стрінгу, породжується виключення out_of _range.

Для роботи зі стрінгами цілком цих операцій досить, а для обробки частин стрінгів (наприклад, пошуку підстрінгу, вставки в стрінг, видалення символів) в класі string визначено безліч різноманітних методів (функцій).

13.3. Функції

Функції класу string для зручності розгляду можна розбити на кілька категорій: присвоювання і додавання частин стрінгів, перетворення стрінгів, пошук підстрінгів, порівняння та отримання характеристик стрінгів.

13.3.1. Присвоєння і додавання частин стрінгів

Для присвоювання частини одного стрінгу іншому служить функція assign:

```
assign (const string & str):
assign (const string & str.size_type pos.size_type n);
assign (const char * s.size_type n);
```

Перша форма функції присвоює стрінг str викликаючому стрінгу, при цьому дія функції еквівалентно операції присвоювання:

```
string s1 ( "Вася"). s2;
s2.assign (s1);        // рівносильно s2 = s1;
```

Друга форма присвоює викликаючому стрінгу частину стрінгу str, починаючи з позиції pos. Якщо pos більше довжини стрінгу, породжується виключення out of range. Викликаючому стрінгу присвоюється n символів, або, якщо pos + n більше, ніж довжина стрінгу str, всі символи до кінця стрінгу str.

Третя форма присвоює викликаючому стрінгу n символів стрінгу s старого типу.

Для додавання частини одного стрінгу до іншого служить функція append:

```
append (const string & str);
append (const string & str. size_type pos. size_type n);
append (const char * s. size_type n);
```

Перша форма функції додає стрінг str до кінця викликаючого стрінгу, при цьому дія функції: еквівалентно операції конкатенації (+).

Друга форма додає до викликаючого стрінгу частину стрінгу str, починаючи з позиції pos. Якщо pos більше довжини стрінгу, породжується виняток out_of_range. До викликаючого стрінгу додається n символів, або, якщо pos + n більше, ніж довжина стрінгу str, всі символи до кінця стрінгу str. Якщо довжина результату більше максимально допустимої довжини стрінгу, породжується виключення length_error.

Третя форма додає до викликаючого стрінгу n символів стрінгу s старого типу.

13.3.2. Перетворення стрінгів

Для вставки в один стрінг частини іншого стрінгу служить функція insert:

```
insert (size_type pos1, const string & str);
insert (size_type pos1, const string & str, size_type pos2, size_type n);
insert (size_type pos, const char * s, size_type n);
```

Перша форма функції вставляє стрінг str в викликаючий стрінг, починаючи з позиції pos1 викликаючого стрінгу. Іншими словами, викликаючий стрінг заміщається стрінгом, який складається з перших pos1 символів викликаючого стрінгу, за якими прямує стрінг str цілком, а після нього розташовуються інші символи викликаючого стрінгу. Якщо pos1 більше довжини стрінгу, породжується виключення out_of_range. Якщо довжина результату більше максимально допустимої довжини стрінгу, породжується виключення length_error.

Друга форма функції вставляє в викликаючий стрінг частину стрінгу str, починаючи з позиції pos1 викликаючого стрінгу. Викликаючий стрінг заміщається стрінгом, який складається з перших pos1 символів викликаючого стрінгу, за якими прямують n елементів стрінгу str, починаючи з позиції pos2, а після них розташовуються інші символи викликаючого стрінгу. Якщо n більше довжини стрінгу str, копіюється весь залишок стрінгу str. Якщо pos1 або pos2 більше довжини відповідного стрінгу, породжується виключення out_of_range.

Якщо довжина результату більше максимально допустимої довжини стрінгу, породжується виключення length_error.

Третя форма функції вставляє в викликаючий стрінг n елементів стрінгу s старого типу, починаючи з позиції pos викликає стрінги.

Для видалення частини стрінга служить функція erase:

```
erase (size_type pos = 0. size_type n = npos):
```

Вона видаляє з викликаючого стрінгу n елементів, починаючи з позиції pos. Якщо pos не вказано, елементи видаляються з початку стрінгу. Якщо не вказано n, видаляється весь залишок стрінгу.

Величина npos є статичним членом класу string і являє собою саме велике додатне число типу size_type (всі одиниці в бітовому поданні).

Очищення всього стрінгу можна виконати за допомогою функції clear:

```
void clear ();
```

Для заміни частини стрінгу служить функція replace:

```
replace (size_type posl, size_type n1, const string & str);
replace (size_type posl, size_type n1, const string & str, size_type pos2,
        size_type n2);
```

Тут pos1 - позиція у викликаючому стрінгу, починаючи з якої виконується заміна, n1 - кількість елементів, що видаляються, pos2 - позиція стрінгу str, починаючи з якої він вставляється в викликаючий стрінг, n2 - кількість елементів, що вставляються, стрінгу str. Якщо pos1 або pos2 більше довжини відповідного стрінгу, породжується виключення out_of_range. Якщо довжина результату більше максимально допустимої довжини стрінгу, породжується виняток length_error.

Третя форма функції заміни дозволяє замінити n1 символів викликаючого стрінгу на n2 символів стрінгу старого стилю s:

```
replace (size_type pos1, size_type nl, const char * s, size_type n2);
```

Для обміну вмісту двох стрінгів служить функція swap:

```
swap (string & s);
```

Для виділення частини стрінгу служить функція substr:

```
string substr (size_type pos = 0, size_type n = npos) const;
```

Ця функція повертає підстрінг викликаючого стрінгу довжиною n, починаючи з позиції pos. Якщо pos більше довжини стрінгу, породжується виключення out_of_range. Якщо n більше довжини стрінгу, повертається весь залишок стрінгу.

Іноді потрібно перетворювати об'єкти типу string в стрінги старого стилю. Для цього призначена функція c_str:

```
const char * c_str () const;
```

Вона повертає константний показник на стрінг, що закінчується нуль-символом. Цей стрінг не можна намагатися змінити. Показник, який на неї посилається, може стати некоректним після будь-якої операції над стрінгом-джерелом.

Аналогічно працює функція data, за тим винятком, що не додає в кінець стрінгу нуль-символ:

```
const char * data () const;
```

Функція copy копіює в масив s n елементів визиваючого стрінгу, починаючи з позиції pos. Нуль-символ в результуючий масив не заноситься. Функція повертає кількість скопійованих елементів:

```
size_type copy (char * s. size_type n. size_type pos = 0) const;
```

Приклад використання функцій зміни вмісту стрінгів:

```
#include <string>
#include <iostream>
using namespace std;
int main () {
string s1 ( "прекрасна королева"), s2 ( "лі"), s3 ( "корова");
cout << "s1 = << s1 << endl;
cout << "s2 =" << s2 << endl;
cout << "s3 =" << s3 << endl;
// Застосування функції insert:
cout << "після insert:" << endl;
cout << "s3 =" << s3.insert (4, s2) << endl;
cout << "s3 =" << s3.insert (7, "к") << endl;
// Застосування функції erase:
s1.erase (0,3):
cout << "після erase:" << endl;
```

```
cout << "s1 =" << s1.erase (11,2) << endl;
// Застосування функції replace:
cout << "після replace:" << endl;
cout << "s1 =" << s1.replace (0,3, s3, 4,2) << endl;
```

Результат роботи програми:

```
s1 = прекрасна королева
s2 = лі
s3 = корова

після insert:
s3 = короліва
s3 = королівка

після erase:
s1 = красна корова

після replace:
s1 = лісна корова
```

13.3.3. Пошук підстрінгів

Для пошуку в класі string передбачено велике розмаїття функцій. Нижче приведені основні:

```
size_type find (const string & str, size_type pos = 0) const;
```

Шукає найлівіше входження стрінгу str в викликаючому стрінгу, починаючи з позиції pos, і повертає позицію стрінгу або npos, якщо стрінг не знайдено.

```
size_type find (char c, size_type pos = 0) const;
```

Шукає найлівіше входження символу c в викликаючому стрінгу, починаючи з позиції pos, і повертає позицію символу або npos, якщо символ не знайдено.

```
size_type rfind (const string & str, size_type pos = npos) const;
```

Шукає саме праве входження стрінгу str в викликаючий стрінг, до позиції pos, і повертає позицію стрінгу або npos, якщо стрінг не знайдено.

```
size_type rfind (char c, size_type pos = npos) const;
```

Шукає саме праве входження символу c в викликаючий стрінг, до позиції pos, і повертає позицію символу або npos, якщо символ не знайдено.

size_type find_first_of (const string & str, size_type pos = 0) const;

Шукає найлівіше входження будь-якого символу стрінга str в викликаючий стрінг, починаючи з позиції pos, і повертає позицію символу або npos, якщо входження не знайдено.

size_type find_first_of (char c, size_type pos = 0) const;

Шукає найлівіше входження символу c в викликаючий стрінг, починаючи з позиції pos, і повертає позицію символу або npos, якщо входження не знайдено.

size_type find_last_of (const string & str, size_type pos = npos) const;

Шукає саме праве входження будь-якого символу стрінгу str в викликаючому стрінгу, починаючи з позиції pos, і повертає позицію символу або npos, якщо входження не знайдено.

size_type find_last_of (char c, size_type pos = npos) const;

Шукає саме праве входження символу c в викликаючому стрінгу, починаючи з позиції pos, і повертає позицію символу або npos, якщо входження не знайдено.

size_type find_first_not_of (const string & str, size_type pos = 0) const;

Шукає найлівішу позицію, починаючи з позиції pos, для якої жоден символ стрінгу str не збігається з символом викликаючого стрінгу.

size_type find_first_not_of (char c, size_type pos = 0) const;

Шукає найлівішу позицію, починаючи з позиції pos, для якої символ c не збігається з символом визиваючого стрінгу.

size_type find_last_not_of (const string & str, sizetype pos = npos) const;

Шукає найправішу позицію до позиції pos, для якої жоден символ стрінгу str не збігається з символом викликаючого стрінгу.

size_type find_last_not_of (char c, size_type pos = npos) const;

Шукає найправішу позицію до позиції pos, для якої символ c не збігається з символом викликаючого стрінгу.

Для кожної функції існує варіант, що дозволяє шукати в заданому стрінгу підстрінг старого стилю.

Приклад застосування функцій пошуку:

```
#include <string>
#include <iostream>
using namespace std;
int main () {
string s1 ( "лісова королівa"), s2 ("лі");
cout << "s1 =" << s1 << endl;
cout << "s2 =" << s2 << endl;
int i = s1.find (s2);
int j = s1.rfind (s2);
cout << "перше s2 в s1" << i << endl;
cout << "останнє s2 в s1" << j << endl;
cout << "перше 'o' в s1" << s1.find ( 'o') << endl;
cout << "останнє 'o' в s1" << s1.rfind ( 'o') << endl;
cout << "перше в s1" << s1.find_first_of ( "абвгді") << endl;
cout << "останнє в s1" << s1.find_last_of ( "абвгді") << endl;
```

Результат роботи програми:

```
S1 = лісова королева
s2 = лі
перше s2 в s1   0
останнє s2 в s1 11
Перше 'o' в s1 3
останнє 'o' в s1 8
перше в s1 1
останнє в s1 14
```

13.3.4. Порівняння частин стрінгів

Для порівняння стрінгів цілком застосовуються перевантажені операції відношення, а якщо потрібно порівнювати частини стрінгів, використовується функція compare:

```
int compare (const string & str) const;
int compare (size_type pos1, size_type n1, const stri ng & str) const;
int compare (size_type pos1, size_type n1,
```

```
const string & str, size_type pos2, size_type n2) const;
```

Перша форма функції порівнює два стрінги цілком і повертає значення, менше 0, якщо викликаючий стрінг лексикографічно менше str, рівне нулю, якщо стрінги однакові, і більше нуля - якщо викликаючий стрінг більше. Ця форма є аналогом функції порівняння стрінгів strstr бібліотеці C.

Друга форма функції виконує аналогічні дії, але порівнює зі стрінгом str n1 символів викликаючого стрінгу, починаючи з pos1.

Третя форма функції порівнює n1 символів викликаючого стрінгу, починаючи з pos1, з підстрінгом стрінга str довжиною n2 символів, починаючи з pos2.

Аналогічні форми функцій існують і для порівняння стрінгів типу string зі стрінгами старого стилю.

Приклад використання функції порівняння стрінгів:

```
#inclucle <string>
#inclucle <iostream>
using namespace std:
int main () {
string s1 ( "лісова королева"), s2 ( "лі"), s3 ( "корова");
cout << "s1=" << s1 «endl;
cout << "s2 =" << s2 << endl;
cout << "s3 =" << s3 << endl;
if (s2.compare (s3)> 0) cout << "s2> s3" << endl;
if (s1.compare (7, 4, s3) <0) cout << "s1 [7-10] <s3" << endl;
if (s1.compare (7, 4, s3, 0,4) == 0) cout << "s1 [7-10] == s3 [0-3]" << endl;
}
```

Результат роботи програми:

```
s1 = лісова королева
s2 = лі
s3 = корова
s2> s3
s1 [7-10] <s3
s1 [7-10] == s3 [0-3]
```

13.3.5. Отримання характеристик стрінгів

У класі string визначено кілька функцій-членів, що дозволяють отримати довжину стрінгу і обсяг пам'яті, займаний об'єктом:

```
size_type size () const:          // Кількість елементів стрінгу
size_type length () const:        // Кількість елементів стрінгу
size_type max_size () const:      // Максимальна довжина стрінгу
size_type capacity () const:      // Об'єм пам’яті, що займає стрінг
bool empty () const:              // Істина. якщо стрінг порожній.
```

Лекція 14. Робота з файлами.

14.1. Файли

Зазвичай у нас є набагато більше даних, ніж здатна вмістити основна пам'ять нашого комп'ютера, тому більша частина інформації зберігається на дисках або інших засобах зберігання даних високої ємності. Такі пристрої також запобігають зникнення даних при виключенні комп'ютера - такі дані є персистентними. На самому нижньому рівні файл просто є послідовність байтів, пронумерованих починаючи з нуля.

Файл має формат: інакше кажучи, він має набір правил. що визначають зміст байтів. Наприклад, якщо файл є текстовим, то перші чотири байти представляють собою перші чотири символи. З іншого боку, якщо файл зберігає бінарне представлення цілих чисел, то перші чотири байти використовуються для бінарного представлення першого цілого числа. Формат по відношенню до файлів на диску грає ту ж роль, що і типи по відношенню до об'єктів в основній пам'яті. Ми можемо приписати бітам, записаним у файлі, певний сенс тоді і тільки тоді, коли відомий його формат.

При роботі з файлами потік ostream перетворює об'єкти, що зберігаються в основній пам'яті, в потоки байтів і записує їх на диск. Потік istream діє навпаки: інакше кажучи, він зчитує потік байтів з диска і складає з них об'єкт.

Найчастіше ми припускаємо, що байти на диску є символами зі звичайного набору символів. Це не завжди так, але, оскільки інші уявлення обробити нескладно, ми, як правило. дотримуватимемося цього припущення. Крім того, будемо вважати, що всі файли находяться на дисках (тобто на обертових магнітних пристроях зберігання даних). І знову-таки, це не завжди так (згадайте про флеш-пам'яті), але на даному рівні програмування фактичне пристрій зберігання не має значення. Це одне з головних переваг абстракцій файлу і потоку.

Для того щоб прочитати файл, ми повинні

1. знати його ім'я;

2. відкрити його (для читання);
3. зчитати символи;
4. закрити файл (хоча це зазвичай виконується неявно).

Для того щоб записати файл, ми повинні
1. іменувати його;
2. відкрити файл (для запису) або створити новий файл з таким ім'ям;
3. записати наші об'єкти;
4. закрити файл (хоча це зазвичай виконується неявно).

Ми вже знаємо основи читання і запису, оскільки у всіх розглянутих нами ситуаціях потік ostream, пов'язаний з файлом, поводиться точно так само, як потік cout, а потік istream, пов'язаний з файлом, поводиться точно так само, як об'єкт cin. Операції, характерні тільки для файлів, ми розглянемо пізніше, а поки подивимося, як відкрити файли, і зосередимо свою увагу на операціях і прийомах, які можна застосувати до всіх потоків ostream і istream.

14.2. Огляд механізмів введення-виведення в Linux

У мові C для здійснення файлового введення-виведення використовуються механізми стандартної бібліотеки мови, оголошені в заголовному файлі stdio.h. Як ви незабаром дізнаєтеся консольне введення-виведення - це не більше ніж приватний випадок файлового введення-виведення. У C++ для введення-виведення найчастіше використовуються потокові типи даних. Однак всі ці механізми є всього лише надбудовами над низькорівневими механізмами введення-виведення ядра операційної системи.

З точки зору моделі КІС (Клієнт-Інтерфейс-Сервер), сервером стандартних механізмів введення виведення мови C (printf, scanf, FILE *, fprintf, fputc і т.д.) Є бібліотека мови. А сервером низкоуровневого введення-виведення в Linux, є саме ядро операційної системи.

Призначені для користувача програми взаємодіють з ядром операційної системи за допомогою спеціальних механізмів, які називаються системними викликами (system calls, syscalls). Зовні системні виклики реалізовані у вигляді звичайних функцій мови C, проте кожного разу викликаючи таку функцію, ми звертаємося безпосередньо до ядра операційної системи. Список всіх системних викликів Linux можна знайти в файлі /usr/include/asm/unistd.h. У цьому розділі ми розглянемо основні системні виклики, які здійснюють введення-виведення: open (), close (), read (), write (), lseek () і деякі інші.

Файлові дескриптори.У мові C при здійсненні введення-виведення ми використовуємо покажчик FILE *. Навіть функція printf() в підсумку зводиться до виклику vfprintf (stdout, ...), різновиди функції fprintf(); константа stdout має тип struct _IO_FILE*, синонімом якого є тип FILE*. Це я до того, що консольне введення-виведення - це файловий введення-виведення. Стандартний потік введення, стандартний потік виведення і потік помилок (як в C, так і в C++) - це

файли. У Linux все, куди можна щось записати або звідки можна щось прочитати представлено (або може бути представлено) у вигляді файлу. Екран, клавіатура, апаратні і віртуальні пристрої, канали, сокети - все це файли. Це дуже зручно, оскільки до всього можна застосовувати одні й ті ж механізми введення-виведення, з якими ми і познайомимося в цьому розділі. Володіння механізмами низкорівневого введення-виведення дає свободу переміщення даних в Linux. Робота з локальними файловими системами, межсетевое взаємодія, робота з апаратними пристроями, - все це здійснюється в Linux за допомогою низкоуровневого введення-виведення.

Ви вже знаєте з попередньої глави, що при запуску програми в системі створюється новий процес (тут є свої особливості, про які поки говорити не будемо). У кожного процесу (крім init) є свій батьківський процес (parent process або просто parent), для якого новоспечений процес є дочірнім (child process, child). Кожен процес отримує копію оточення (environment) батьківського процесу. Виявляється, крім оточення дочірній процес отримує в якості багажу ще й копію таблиці файлових дескрипторів.

Файловий дескриптор (File descriptor) - це ціле число (int), відповідне відкритого файлу. Дескриптор, відповідний реально відкритого файлу завжди більше або дорівнює нулю. Копія таблиці дескрипторів (читай: таблиці відкритих файлів всередині процесу) прихована в ядрі. Ми не можемо отримати прямий доступ до цієї таблиці, як при роботі з оточенням через environ. Можна, звичайно, дещо "витягнути" через дерево / proc, але нам це не треба. Програміст повинен лише розуміти, що кожен процес має свою копію таблиці дескрипторів. В межах одного процесу все дескриптори унікальні (навіть якщо вони відповідають одному і тому ж файлу або пристрою). У різних процесах дескриптори можуть збігатися або не збігатися - це не має ніякого значення, оскільки у кожного процесу свій власний набір відкритих файлів.

Виникає питання: скільки файлів може відкрити процес? У кожній системі є свій ліміт, який залежить від конфігурації. Якщо ви використовуєте bash або ksh (Korn Shell), то можете скористатися внутрішньою командою оболонки ulimit, щоб дізнатися це значення.

```
$ Ulimit -n
1024
$
```

Якщо ви працюєте з оболонкою C-shell (csh, tcsh), то у вашому розпорядженні команда limit:

```
$ Limit descriptors
descriptors 1024
$
```

У командній оболонці, в якій ви працюєте (bash, наприклад), відкриті три файли: стандартний ввід (дескриптор 0), стандартний висновок (дескриптор 1) і стандартний потік помилок (дескриптор 2). Коли під оболонкою запускається програма, в системі створюється новий процес, який є для цієї оболонки дочірнім процесом, отже, отримує копію таблиці дескрипторів свого батька (тобто всі відкриті файли батьківського процесу). Таким чином програма може здійснювати консольне введення-виведення через ці дескриптори. Протягом всієї книги ми будемо часто грати з цими дескрипторами.

Таблиця дескрипторів, крім усього іншого, містить інформацію про поточну позиції читання-запису для кожного дескриптора. При відкритті файлу, позиція читання-запису встановлюється в нуль. Кожен прочитаний чи записаний байт збільшує на одиницю покажчик поточної позиції.

Відкриття файлу: системний виклик open (). Щоб отримати можливість прочитати щось з файлу або записати щось в файл, його потрібно відкрити. Це робить системний виклик open(). Цей системний виклик не має постійного списку аргументів (за рахунок використання механізму va_arg); в зв'язку з цим існують дві "різновиди" open(). Не тільки в C ++ є перевантаження функцій ;-) Якщо цікаво, то про механізм va_arg можна прочитати на man-сторінці stdarg (man 3 stdarg) або в книзі Б. Керніган і Д. Рітчі "Мова програмування Сі". Нижче наведені адаптовані прототипи системного виклику open().

```
int open (const char * filename, int flags, mode_t mode);
int open (const char * filename, int flags);
```

Системний виклик open() оголошений в заголовки fcntl.h. Нижче наведено загальний адаптований прототип open().

```
int open (const char * filename, int flags, ...);
```

Почнемо по порядку. Перший аргумент - ім'я файлу в файлової системі в звичайній формі: повний шлях до файлу (якщо файл не знаходиться в поточному каталозі) або скорочене ім'я (якщо файл в поточному каталозі).

Другий аргумент - це режим відкриття файлу, що представляє собою один або кілька прапорів відкриття, об'єднаних оператором побітового АБО. Найбільш часто використовують тільки перші сім прапорів. Якщо ви хочете, наприклад, відкрити файл в режимі читання і запису, і при цьому автоматично створити файл, якщо такого не існує, то другий аргумент open() буде виглядати приблизно так: O_RDWR | O_CREAT. Константи-прапори відкриття оголошені в заголовки bits/fcntl.h, однак не варто включати цей файл в свої програми, оскільки він вже включений в файл fcntl.h.

Третій аргумент використовується в тому випадку, якщо open() створює новий файл. В цьому випадку файлу потрібно задати права доступу (режим), з

якими він з'явиться в файлової системі. Права доступу задаються перерахуванням прапорів, об'єднаних побітовим АБО. Замість прапорів можна використовувати число (як правило вісімкове), проте перший спосіб наочніше і краще. Щоб, наприклад, створений файл був доступний в режимі "читання-запис" користувачем і групою і "тільки читання" іншими користувачами, - в третьому аргументі open() треба вказати приблизно наступне: S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH або 0664. прапори режиму доступу реально оголошені в заголовки bits / stat.h, але він не призначений для включення в призначені для користувача програми, і замість нього ми повинні включати файл sys / stat.h. Тип mode_t оголошений в заголовки sys / types.h.

Якщо файл був успішно відкритий, open() повертає файловий дескриптор, за яким ми будемо звертатися до файлу. Якщо сталася помилка, то open() повертає -1.

Закриття файлу: системний виклик close(). Системний виклик close() закриває файл. Взагалі кажучи, по завершенні процесу всі відкриті файли (крім файлів з дескрипторами 0, 1 і 2) автоматично закриваються. Тим не менш, це не звільняє нас від самостійного виклику close(), коли файл потрібно закрити. До того ж, якщо файли не закривати самостійно, то відповідні дескриптори не звільняються, що може привести до перевищення ліміту відкритих файлів. Простий приклад: додаток може бути налаштований так, щоб кожну хвилину відкривати і перечитувати свій файл конфігурації для перевірки оновлень. Якщо кожен раз файл не буде закриватися, то в моїй системі, наприклад, додаток може "накритися мідним тазом" приблизно через 17 годин. Автоматично! Крім того, файлова система Linux підтримує механізм буферизації. Це означає, що дані, які нібито записуються, реально записуються на носій (синхронізуються) тільки через якийсь час, коли система вважатиме це правильним і оптимальним. Це підвищує продуктивність системи і навіть продовжує ресурс жорстких дисків. Системний виклик close() не форсує запис даних на диск, однак дає більше гарантій того, що дані залишаться в цілості й схоронності.

Системний виклик close() оголошений в файлі unistd.h. Нижче наведено його адаптований прототип.

```
int close (int fd);
```

Очевидно, що єдиний аргумент - це файловий дескриптор. Значення, що повертається - нуль в разі успіху, і -1 - у разі помилки. Досить часто close() викликають без перевірки значення, що повертається. Це не дуже груба помилка, але, тим не менш, іноді закриття файлу буває невдалим (у разі неправильного дескриптора, в разі переривання функції за сигналом або в разі помилки введення-виведення, наприклад). У будь-якому випадку, якщо програма повідомить користувачеві, що файл неможливо закрити, це добре.

Тепер можна написати простінків програму, яка використовує системні виклики open() і close(). Ми ще не вміємо читати з файлів і писати в файли, тому напишемо програму, яка створює файл з ім'ям, переданим в якості аргументу (argv [1]) і з правами доступу 0600 (читання і запис для користувача). Нижче наведено вихідний код програми.

```
/ * Openclose.c * /
#include <fcntl.h>/ * Open () and O_XXX flags * /
#include <sys / stat.h>      / * S_IXXX flags * /
#include <sys / types.h>    / * Mode_t * /
#include <unistd.h>          / * Close () * /
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char ** argv)
{
  int fd;
  mode_t mode = S_IRUSR | S_IWUSR;
  int flags = O_WRONLY | O_CREAT | O_EXCL;
  if (argc <2)
  {
            fprintf (stderr, "openclose: Too few arguments \ n");
            fprintf (stderr, "Usage: openclose <filename> \ n");
            exit (1);
  }

  fd = open (argv [1], flags, mode);
  if (fd <0)
  {
            fprintf (stderr, "openclose: Can not open file '% s' \ n",
                              argv [1]);
            exit (1);
  }

  if (close (fd)! = 0)
  {
            fprintf (stderr, "Can not close file (descriptor =% d) \ n", fd);
            exit (1);
  }
  exit (0);
}
```

Зверніть увагу, якщо запустити програму двічі з одним і тим же аргументом, то на другий раз open() видасть помилку. У цьому винен прапор O_EXCL, який "дає добро" тільки на створення ще не існуючих файлів. Наочності заради, прапори відкриття і прапори режиму ми занесли в окремі змінні, проте можна було б зробити так:

```
fd = open (argv [1], O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
```

Або так:

```
fd = open (argv [1], O_WRONLY | O_CREAT | O_EXCL, 0600);
```

Читання файлу: системний виклик read (). Системний виклик read(), оголошений у файлі unistd.h, дозволяє читати дані з файлу. На відміну від бібліотечних функцій файлового введення-виведення, які надають можливість інтерпретації зчитувальних даних. Можна, наприклад, записати в файл наступне вміст:

```
2006
```

Тепер, використовуючи бібліотечні механізми, можна читати файл по-різному:

```
fscanf (filep, "% s", buffer);
fscanf (filep, "% d", number);
```

Системний виклик read() читає дані в "сирому" вигляді, тобто як сукупність електронних даних, без будь-якої інтерпретації. Нижче представлений адаптований прототип read().

```
ssize_t read (int fd, void * buffer, size_t count);
```

Перший аргумент - це файловий дескриптор. Тут більше сказати нічого. Другий аргумент - це покажчик на область пам'яті, куди будуть поміщатися дані. Третій аргумент - кількість байт, які функція read() буде намагатися прочитати з файлу. Значення, що повертається - кількість прочитаних байт, якщо читання відбулося і -1, якщо сталася помилка. Хочу зауважити, що якщо read() повертає значення менше count, то це не символізує про помилку.

Хочу сказати кілька слів про типи. Тип size_t в Linux використовується для зберігання розмірів блоків пам'яті. Який тип реально ховається за size_t, залежить від архітектури; як правило це unsigned long int або unsigned int. Тип ssize_t (Signed SIZE Type) - це той же size_t, тільки знаковий. Використовується,

наприклад, в тих випадках, коли потрібно повідомити про помилку, повернувши негативний розмір блоку пам'яті. Системний виклик read() саме так і поступає.

Тепер напишемо програму, яка просто читає файл і виводить його вміст на екран. Файл буде передаватися в якості аргументу (argv [1]). Нижче наведено вихідний код цієї програми.

```
/ * Myread.c * /
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys / stat.h>
#include <sys / types.h>

int main (int argc, char ** argv)
{
	int fd;
	ssize_t ret;
	char ch;
	if (argc <2)
	{
		fprintf (stderr, "Too few arguments \ n");
		exit (1);
	}

	fd = open (argv [1], O_RDONLY);
	if (fd <0)
	{
		fprintf (stderr, "Can not open file \ n");
		exit (1);
	}

	while ((ret = read (fd, & ch, 1))> 0)
	{
		putchar (ch);
	}

	if (ret <0)
	{
		fprintf (stderr, "myread: Can not read file \ n");
		exit (1);
	}
```

```
        close (fd);
        exit (0);
    }
```

У цьому прикладі використовується укорочена версія open(), так як файл відкривається тільки для читання. Як буфера (другий аргумент read()) ми передаємо адресу змінної типу char. За цією адресою будуть зчитуватися дані з файлу (по одному байту за раз) і передаватися на стандартний висновок. Цикл читання файлу закінчується, коли read() повертає нуль (нічого більше читати) або -1 (помилка). Системний виклик close() закриває файл.

Як можна помітити, в нашому прикладі системний виклик read() викликається рівно стільки разів, скільки байт міститься у файлі. Іноді це дійсно потрібно; але не тут. Читання-запис посимвольним методом (як в нашому прикладі) значно уповільнює процес введення-виведення за рахунок багаторазових звернень до системних викликів. З цієї ж причини зростає ймовірність виникнення помилки. Якщо немає дійсної необхідності, файли потрібно читати блоками. Про те, який розмір блоку краще, буде розказано в наступних розділах книги. Нижче наведено вихідний код програми, яка робить те ж саме, що і попередній приклад, але з використанням блочного читання файлу. Розмір блоку встановлений в 64 байта.

```
/ * Myread1.c * /
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys / stat.h>
#include <sys / types.h>

#define BUFFER_SIZE       64

int main (int argc, char ** argv)
{
  int fd;
  ssize_t read_bytes;
  char buffer [BUFFER_SIZE + 1];
  if (argc <2)
  {
        fprintf (stderr, "Too few arguments \ n");
        exit (1);
  }
```

```
    fd = open (argv [1], O_RDONLY);
    if (fd <0)
    {
            fprintf (stderr, "Can not open file \ n");
            exit (1);
    }

    while ((read_bytes = read (fd, buffer, BUFFER_SIZE))> 0)
    {
            buffer [read_bytes] = 0; / * Null-terminator for C-string * /
            fputs (buffer, stdout);
    }

    if (read_bytes <0)
    {
            fprintf (stderr, "myread: Can not read file \ n");
            exit (1);
    }
    close (fd);
    exit (0);
}
```

Тепер можна приблизно оцінити і порівняти швидкість роботи двох прикладів. Для цього треба вибрати в системі досить великий файл (бінарник ядра або відеофільм, наприклад) і подивитися на те, як швидко читаються ці файли:

```
$ Time ./myread / boot / vmlinuz> / dev / null

real 0m1.443s
user 0m0.383s
sys 0m1.039s
$ Time ./myread1 / boot / vmlinuz> / dev / null

real 0m0.055s
user 0m0.010s
sys 0m0.023s
$
```

Запис в файл: системний виклик write(). Для запису даних у файл використовується системний виклик write(). Нижче представлений його прототип.

ssize_t write (int fd, const void * buffer, size_t count);

Як бачите, прототип write() відрізняється від read() тільки специфікатором const в другому аргументі. В принципі write() виконує процедуру, зворотну read (): записує count байтів з буфера buffer в файл з дескриптором fd, повертаючи кількість записаних байтів або -1 у випадку помилки. Так просто, що можна відразу переходити до прикладу.

```
/ * Rw.c * /
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>          / * Read (), write (), close () * /
#include <fcntl.h>/ * Open (), O_RDONLY * /
#include <sys / stat.h>      / * S_IRUSR * /
#include <sys / types.h>     / * Mode_t * /

#define BUFFER_SIZE        64

int main (int argc, char ** argv)
{
  int fd;
  ssize_t read_bytes;
  ssize_t written_bytes;
  char buffer [BUFFER_SIZE];
  if (argc <2)
  {
            fprintf (stderr, "Too few arguments \ n");
            exit (1);
  }

  fd = open (argv [1], O_RDONLY);
  if (fd <0)
  {
            fprintf (stderr, "Can not open file \ n");
            exit (1);
  }

  while ((read_bytes = read (fd, buffer, BUFFER_SIZE))> 0)
  {
            / * 1 == stdout * /
            written_bytes = write (1, buffer, read_bytes);
```

```
            if (written_bytes! = read_bytes)
            {
                fprintf (stderr, "Can not write \ n");
                exit (1);
            }
    }

    if (read_bytes <0)
    {
            fprintf (stderr, "myread: Can not read file \ n");
            exit (1);
    }
    close (fd);
    exit (0);
}
```

У цьому прикладі нам уже не треба ізощеряться в спробах вставити нуль-термінатор в рядок для запису, оскільки системний виклик write() НЕ запише більшу кількість байт, ніж ми йому вказали. В даному випадку для демонстрації write() ми просто записували дані в файл з дескриптором 1, тобто в стандартний висновок. Але перш, ніж переходити до читання наступного розділу, спробуйте самостійно записати що-небудь (за допомогою write(), природно) в звичайний файл. Коли будете відкривати файл для запису, зверніть будь ласка увагу на прапори O_TRUNC, O_CREAT і O_APPEND. Подумайте, чи всі прапори поєднуються між собою за змістом.

Довільний доступ: системний виклик lseek (). Як вже говорилося, з кожним відкритим файлом пов'язано число, яке вказує на поточну позицію читання-запису. При відкритті файлу позиція дорівнює нулю. Кожен виклик read() або write() збільшує поточну позицію на значення, яка дорівнює кількості прочитаних або записаних байт. Завдяки цьому механізму, кожен повторний виклик read() читає такі дані, і кожен повторний write() записує дані в продовження попередніх, а не затирає старі. Такий механізм послідовного доступу дуже зручний, проте іноді потрібно отримати довільний доступ до вмісту файлу, щоб, наприклад, прочитати або записати файл заново.

Для зміни поточної позиції читання-запису використовується системний виклик lseek(). Нижче представлений його прототип.

```
off_t lseek (int fd, ott_t offset, int against);
```

Перший аргумент, як завжди, - файловий дескриптор. Другий аргумент - зміщення, як позитивне (вперед), так і негативне (тому). Третій аргумент зазвичай передається у вигляді однієї з трьох констант SEEK_SET, SEEK_CUR і

SEEK_END, які показують, від якого місця відраховується зміщення. SEEK_SET - означає початок файлу, SEEK_CUR - поточна позиція, SEEK_END - кінець файлу. Розглянемо наступні виклики:

```
lseek (fd, 0, SEEK_SET);
lseek (fd, 20, SEEK_CUR);
lseek (fd, -10, SEEK_END);
```

Перший виклик встановлює поточну позицію в початок файлу. Другий виклик зміщує позицію вперед на 20 байт. У третьому випадку поточна позиція переміщається на 10 байт назад щодо кінця файлу.

У разі вдалого завершення, lseek() повертає значення встановленої "нової" позиції щодо початку файлу. У разі помилки повертається -1.

Я довго думав, який би приклад придумати, щоб продемонструвати роботу lseek() наочним чином. Найбільш підходящим прикладом мені здалася ідея створення програми малювання символами. Програма виявилася не надто простий, однак якщо ви зможете розібратися в ній, то можете вважати, що успішно оволоділи азами низкоуровневого введення-виведення Linux. Нижче представлений вихідний код цієї програми.

```
/ * Draw.c * /
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys / types.h>
#include <sys / stat.h>
#include <string.h>          / * Memset () * /

#define N_ROWS  15         / * Image height * /
#define N_COLS   40         / * Image width * /
#define FG_CHAR 'O'         / * Foreground character * /
#define IMG_FN   "Image"/ * Image filename * /

#define N_MIN (A, B) ((A) <(B)? (A) :( B))
#define N_MAX (A, B) ((A)> (B)? (A) :( B))

static char buffer [N_COLS];

void init_draw (int fd)
{
  ssize_t bytes_written = 0;
```

```
  memset (buffer, '', N_COLS);
  buffer [N_COLS] = '\ n';
  while (bytes_written <(N_ROWS * (N_COLS + 1)))
          bytes_written + = write (fd, buffer, N_COLS + 1);
}

void draw_point (int fd, int x, int y)
{
  char ch = FG_CHAR;
  lseek (fd, y * (N_COLS + 1) + x, SEEK_SET);
  write (fd, & ch, 1);
}

void draw_hline (int fd, int y, int x1, int x2)
{
  size_t bytes_write = abs (x2-x1) + 1;
  memset (buffer, FG_CHAR, bytes_write);
  lseek (fd, y * (N_COLS + 1) + N_MIN (x1, x2), SEEK_SET);
  write (fd, buffer, bytes_write);
}

void draw_vline (int fd, int x, int y1, int y2)
{
  int i = N_MIN (y1, y2);
  while (i <= N_MAX (y2, y1)) draw_point (fd, x, i ++);
}

int main (void)
{
  int a, b, c, i = 0;
  char ch;
  int fd = open (IMG_FN, O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd <0) {
          fprintf (stderr, "Can not open file \ n");
          exit (1);
  }

  init_draw (fd);
  char * icode [] = { "v 1 + 1 11", "v 11 7 11", "v 14 5 11",
  "V 18 6 11", "v 21 5 10", "v 25 5 10", "v 29 5 6", "v 33 5 6",
  "V 29 10 11", "v 33 10 11", "h 11 1 8", "h 5 16 17",
```

```
        "H 11 22 24", "p 11 5 0 Попереднє", "p 15 6 0", "p 26 11 0
Попереднє", "p 30 Разом 7 0",
        "P 32 7 0", "p 31 8 0", "p 30 9 0", "p 32 9 0", NULL};

        while (icode [i]! = NULL) {
        sscanf (icode [i], "% c% d% d% d", & ch, & a, & b, & c);
                switch (ch) {
                        case 'v': draw_vline (fd, a, b, c); break;
                        case 'h': draw_hline (fd, a, b, c); break;
                        case 'p': draw_point (fd, a, b); break;
                        default: abort ();
                }
                i ++;
        }
        close (fd);
        exit (0);
}
```

Тепер розберемося, як працює ця програма. Спочатку "полотно" заповнюється пробілами. Функція init_draw() через підрядник записує в файл прогалини, щоб вийшов "полотно", розміром N_ROWS на N_COLS. Масив рядків icode в функції main() - це набір команд малювання. Команда починається з однією з трьох літер: 'v' - намалювати вертикальну лінію, 'h' - намалювати горизонтальну лінію, 'p' - намалювати точку. Після кожної такої літери слідують три числа. У разі вертикальної лінії перше число - фіксована координата X, а два інших числа - це початкова і кінцева координати Y. У випадку горизонтальної лінії фіксується координата Y (перше число). Два інших числа - початкова координата X і кінцева координата X. При малюванні точки використовуються тільки два перших числа: координата X і координата Y. Отже, функція draw_vline() малює вертикальну лінію, функція draw_hline() малює горизонтальну лінію, а draw_point () малює точку.

Функція init_draw() пише у файл N_ROWS рядків, кожна з яких містить N_COLS прогалин, що закінчуються перекладом рядка. Це процедура підготовки "полотна".

Функція draw_point() обчислює позицію (виходячи з значень координат), переміщує туди поточну позицію введення-виведення файлу, і записує в цю позицію символ (FG_CHAR), яким ми малюємо "картину".

Функція draw_hline() заповнює частину рядка символами FG_CHAR. Так виходить горизонтальна лінія. Функція draw_vline() працює інакше. Щоб записати вертикальну лінію, потрібно записувати по одному символу і кожен раз "перескакувати" на наступний рядок. Ця функція працює повільніше, ніж draw_hline(), але інакше ми не можемо.

Отримане зображення записується в файл image. Будьте уважні: щоб розвантажити вихідний код, з програми виключені багато перевірки (read(), write(), close(), діапазон координат та ін.). Спробуйте включити ці перевірки самостійно.

14.3. Технології роботи з потоками файлів

Відкриття файлу. Якщо ви хочете зчитати дані з файлу або записати їх у файл, відкрийте потік конкретно для цього файлу. Потік ifstream - це потік istream для читання з файлу, потік ofstream - це потік ostream для запису в файл, а потік fstream є потік iostream, який можна використовувати як для читання, так і для запису. Перед використанням файловий потік повинен бути пов'язаний з файлом, наприклад:

```
cout << "Будь ласка, введіть ім'я файлу:";
string iname; cin >> iname;
ifstream ist {iname};        // ist - вхідний потік для файлу iname
if (!ist) error ( "Неможливо відкрити вхідний файл", name);
```

Визначення потоку ifstream з ім'ям, заданим рядком name, відкриває файл з цим ім'ям для читання. Перевірка !ist дозволяє з'ясувати, чи був файл відкритий коректно. Після цього можна зчитувати дані з файлу точно так же. як з будь-якого іншого потоку istream. Наприклад, в припущенні, що оператор введення >> визначено для типу Point, можна написати наступний фрагмент програми:

```
vector <Point> points;
for (Point p; ist >> p;)
        points. push_back (p);
```

Виведення в файли виконується аналогічно, за допомогою потоків ofstream. Розглянемо приклад.

```
cout << "Введіть ім’я файлу для виведення:";
string oname;
cin >> oname;
ofstream ost {onaше}; // ost - вихідний потік для файлу oname
if (!ost) error ( "неможливий відкрити вхідний файл", oname);
```

Визначення потоку ofstream з ім'ям, заданим рядком name, відкриває файл з цим ім'ям для запису. Перевірка !ost дозволяє з'ясувати, чи був файл успішно відкритий. Після цього можна записувати дані в файл точно так же, як і в будь-який інший потік ostream. Наприклад:

```
for (int p: points)
ost << '(' << p.x << ',' << p.y << ") \ n";
```

Коли файловий потік виходить з області видимості, пов'язаний з ним файл закривається. Коли файл закривається, пов'язаний з ним буфер зкидується; інакше кажучи, символи з буфера записуються в файл.

Як правило, файли в програмі найкраще відкривати якомога раніше, до виконання будь-яких серйозних обчислень. Зрештою, було б занадто марнотратним виконати більшу частину роботи і виявити, що ви не можете її завершити, тому що вам нікуди записати результати.

В ідеалі відкриття файлу виконується неявно як частина процесу створення потоків ostream і istream, а його закриття ґрунтується на області видимості потоку, наприклад:

```
void fill_from_file (vector <Point> & points, strinq & name)
{
ifstream ist {name};        // Відкриття файла для читання
if (! ist) error ( "неможливо відкрити вхідний файл", name);
//. . . використання ist. . .
// Неявне закриття файлу при виході з функції
```

Операції open() і close() можуть бути виконані і явно. Однак орієнтація на область видимості мінімізує шанси. що хтось спробує використовувати файловий потік до того, як файл буде пов'язаний з потоком. або після його закриття. наприклад:

```
ifstream ifs;
//. . .
ifs >> foo; // Невдало: немає пов'язаного з потоком файлу
//. . .
ifs. open (name,                    // Відкриваємо файл   name. . .
      ios_base:: in);               //. . . для читання
ifs.close ();                       // Закриваємо файл
// ...
ifs >> bar;             // Невдало: пов'язаний з потоком файл закритий
//. . .
```

У реальній програмі виникають проблеми, як правило, набагато складніші. На щастя, ми не можемо відкрити файловий потік вдруге, попередньо його не закривши, наприклад:

```
fstream fs;
fs.open ( "foo", ios_base :: in); // Відкриваємо файл для введення
```

```
// Некоректна функція close ()
fs.open ( "foo", ios_base :: out); // Невдало: потік ifs вже відкрито
if (! fs) error ( "Неможливо");
```

Не забувайте перевіряти потік після його відкриття.

Чому допускається явне використання функцій open() і close()? Справа в тому, що іноді час життя з'єднання потоку з файлом не збігається з його областю видимості. Однак ця подія відбувається так рідко, що про нього можна не турбуватися. Слід також зазначити, що такий код можна зустріти у програмістів, які використовують стилі мов і бібліотек, які не мають ідіоми області видимості, використовуваної потоками iostream (і іншою частиною стандартної бібліотеки C ++).

Читання і запис файлу. Подивимося, як можна було б вважати результати деяких вимірювань з файлу і представити їх в пам'яті. Припустимо, в файлі записана температура повітря, виміряна на метеостанції.

```
0 60.7
1 60.6
2 60.3
3 59.22
...
```

Цей файл містить послідовність пар (годину, температура). Годинники пронумеровані від 0 до 23, а температура виміряна за шкалою Фаренгейта. Подальше форматування не передбачено: інакше кажучи, файл не містить ніяких заголовків (наприклад, інформації про те, де знято показання температури), одиниць вимірювань, знаків пунктуації (наприклад, дужок навколо кожної пари значень) або ознаки кінця файлу. Це найпростіший варіант.

Можна уявити зчитується температуру у вигляді типу Reading.

```
struct Readinq {            // Дані про темпера турі
int      hour;              // Годинники після півночі [0, 23}
double temperature;         // За Фаренгейтом
```

При цьому дані можна зчитувати наступним чином:

```
vector <Reading> temps; // Тут зберігається зчитана інформація
int hour;
double temperature;
while (ist >> hour >> temperature) {
if (hour <0 || 23 <hour) error ( "Некоректне час");
temps.push_back (Reading {hour, temperature});
```

Це типовий цикл введення. Потік istream з ім'ям ist може бути вхідним файловим потоком (ifstream), як в попередньому розділі, стандартним потоком введення (cin) або будь-яким іншим потоком istream. Для коду, подібного до наведеного вище, не має значення, звідки потік istream отримує дані. Все, що потрібно знати нашій програмі, - це те, що потік ist є istream і що дані мають очікуваний формат. Наступний розділ присвячений цікавого питання: як виявляти помилки у вхідних даних і що можна зробити після виявлення помилки форматування.

Записати дані в файл зазвичай простіше, ніж вважати їх звідти. Як і раніше, після ініціалізації потоку ми не зобов'язані знати, що саме він собою являє. Зокрема, ми можемо використовувати вихідний файловий потік (ofstream) з попереднього розділу нарівні з будь-яким іншим потоком ostream. При виведенні ми, наприклад, могли б побажати, щоб кожна пара значень була укладена в дужки.

```
for (int i = 0; i <temps.size (); ++ i)
ost << '(' << temps [i] .hour << ','
<< temps [i].temperature << ") \ n";
```

В результаті ми отримуємо програму, яка читає вихідні дані з файлу і створює новий файл у форматі (годину, температура).

Оскільки файлові потоки автоматично закривають свої файли при виході з області видимості, повна програма приймає наступний вигляд.

```
#include "std_lib_facilities .h"

struct Reading {                        // Дані про температуру
int      hour;                          // Час після опівночі [0, 23]
double temperature;                     // За Фаренгейтом
};

int main ()
{
cout << "Введіть ім'я вхідного файпа:";
string iname;
cin >> iname;
ifstream ist {iname};                   // ist читає дані з файлу iname
if (!ist) error ( "Неможливо відкрити вхідний файл", iname);

string oname;
cout << "Введіть ім'я вихідного файлу:";
```

```
cin >> oname;
ofstream ost {oname};     // ost записує дані в файл proname
if (! ost) error ( "Неможливо відкрити вихідний файл", oname);

vector <Readinq> temps;          // Сховище даних
int hour;
double temperature;
while (ist >> hour >> temperature) {
if (hour <0 || 23 <hour) error ( "Некоректне час");
temps.push_back (Reading {hour, temperature));
for (int i = 0; i <temps.size (); ++ i)
ost << '(' << temps [i] .hour << ','
<< temps [i]. temperature << ") \ n";
```

14.4. Читання структурованого файлу

Припустимо, у файлі записані результати вимірювання температури, мають певну структуру.

- У файлі записані роки, протягом яких проводилися вимірювання.
- Запис про рік починається символами {year, за якими слід ціле число, що позначає рік, наприклад 1900 і закінчується символом}.
- Рік складається з місяців, протягом яких проводилися вимірювання.
- Запис про місяць починається символами {month, за якими слідує три літери від назви місяця, наприклад jan, і закінчується символом}.
- Дані містять свідчення часу і температури.
- Показання починаються з символу (, за якими слідує день місяця, година дня і температура, і закінчуються символом).

Розглянемо приклад.

```
{Year 1990}
{Year тисячі дев'ятсот дев'яносто одна {month jun}}
{Year тисячі дев'ятсот дев'яносто дві {month jan (1 0 61.5)} (month feb (1 1 64) (2 + 2 65 .2)}}
{Year 2000
        {Month feb (1 1 68) (2 3 66.66) (1 0 67.2)}
        {Month dec (15 15 - 9.2) (15 14 -8.8) (14 0 - 2)}
```

Цей формат досить своєрідний. Формати запису файлів взагалі часто виявляються досить специфічними. В цілому в програмної індустрії спостерігається тенденція до широкого використання все більш впорядкованих і ієрархічно структурованих файлів (наприклад, HTML і XML), але в дійсності ми як і раніше рідко можемо управляти форматом файлу, який необхідно прочитати. Файли такі, які вони є, і нам потрібно їх прочитати. Якщо формат занадто

невдалий або файли містять багато помилок, можна написати програму перетворення формату в більш відповідний. З іншого боку, ми, як правило, маємо можливість вибирати уявлення даних в пам'яті в зручному для себе вигляді, а при виборі формату виведення часто керуємося лише власними потребами і смаком.

Припустимо, дані про температуру записані в зазначеному вище фор маті і нам потрібно їх прочитати. На щастя, формат містить автоматично ідентифікуються компоненти, такі як роки і місяці (трохи нагадує формати HTML і XML). З іншого боку, формат окремого запису досить незручний. Наприклад, в ній немає інформації. яка могла б нам допомогти, якби хтось переплутав день місяця з часом або перед ставив температуру за шкалою Цельсія, хоча потрібно було за шкалою Фаренгейта, і навпаки. Нам просто доведеться впоратися з тим, що ми маємо.

14.4.1. Подання в пам'яті

Як уявити ці дані в пам'яті? На перший погляд. необхідно створити три класи, Year, Month і Reading, точно відповідні вхідної інформації. Класи Year і Month очевидним чином могли б виявитися корисними при обробці даних; ми хочемо порівнювати температури різних років, обчислювати середньомісячні температури, порівнювати різні місяці одного року, однакові місяці різних років, показники температури з записами про сонячному випромінюванні і вологості і т.д. В принципі, класи Year і Month точно відображають наші уявлення про температуру і погоду: клас Month містить щомісячну інформацію, а клас Year - щорічну. А як щодо класу Reading? Це поняття низького рівня, пов'язане з частиною апаратного забезпечення (сенсором). Дані в класі Reading (день місяця, годину і температура) мають сенс тільки в рамках класу Month. Крім того, вони не структуровані: ніхто не обіцяв, що дані будуть записані по днях або по годинах. У загальному випадку для того, щоб зробити з даними щось корисне, спочатку їх необхідно впорядкувати.

Для представлення даних про температуру в пам'яті зробимо наступні припущення.

- Якщо є показання для якогось місяця, то їх зазвичай буває багато.
- Якщо є показання для якогось дня, то їх теж зазвичай буває багато.

У цьому випадку доцільно уявити клас Year як вектор, що складається з 12 об'єктів класу Month, клас Month - як вектор, що складається з 30 об'єктів класу Day, а клас Day - як 24 показники температури (по одному на годину). Це дозволяє просто і легко маніпулювати даними при вирішенні найрізноманітніших завдань. Отже, класи Day, Month і Year - це прості структури даних, кожна з яких має конструктор. Оскільки ми плануємо створювати об'єкти класів Month і Day як частина об'єктів класу Year ще до того, як дізнаємося, які показники температури у нас є. то повинні сформулювати, що означає "пропущені дані" для години дня, до зчитування яких ще не дійшла черга.

```
const int not_a_reading = -7777; // Нижче абсолютного нуля
```

Аналогічно ми помітили, що часто протягом декількох місяців не про переводилося жодного виміру, тому ввели поняття "пропущений місяць". замість того щоб перевіряти пропуски для кожного дня.

```
const int not_a_month = -1;
```

Три основні класи приймають такий вигляд:

```
struct Day {
        vector <double> hour {vector <double> (24, not_a_readinq)};
};
```

Інакше кажучи, Day включає 24 години, кожен з яких ініціалізовано значенням not_a_reading.

```
struct Month {                          // Місяць
        int month {not_a_month};        // {0, 11} (січнь відповідає 0)
        vector <Day> day {32};    // {1, 31} по одному вектору на день
```

Ми вирішили пожертвувати одним елементом data [0], щоб спростити код.

```
struct Year {                       // Рік складається з місяців
        int year;                           // Додатне значення
        vector <Month> month {12}; // [0, 11] (січня відповідає 0)
}
```

В принципі, кожен клас - це просто вектор, а класи Month і Year містять ідентифікуючі члени month і year відповідно.

14.4.2. Читання структурованих значень

Клас Reading буде використаний лише для введення даних, і він ще простіше інших:

```
struct Reading {
        int day;
        int hour;
        double temperature;
};

istream & operator >> (istream & is, Reading & r)
// Прочитуємо показники температури з потоку is в r
// Формат: (3 4 9. 7)
// Перевіряємо формат, але не коректність даних
```

```
char ch1;
if (is >> ch1 && ch1! = '(') {// Чи може це бути Reading?
is.unqet ();
is.clear (ios_base :: failbit); return is;
char ch2;
int d;
int h;
double t;
is >> d >> h >> t >> ch2;
if (! is || ch2! = ')')
error ( "Поганий запис");
r. day = d;
r. hour = h;
r. temperature = t;
return is;
}
```

Ми починаємо з перевірки, чи правильно починається формат. Якщо немає, то переводимо файл в стан fail () і виходимо. Це дозволяє нам спробувати вважати інформацію: якось інакше. З іншого боку, якщо помилка формату виявляється після зчитування даних і немає реальних шансів на відновлення роботи, то ми викликаємо функцію error ().

Операція введення: класу Month майже така ж, за винятком того, що в ній зчитується довільне: кількість об'єктів класу Reading, а не фіксований набір значень (: як робить оператор >> в: класі Reading).

```
istream & operator >> (istream & is, Month & m)
// Прочитуємо об'єкт класу Month з потоку is в об'єкт m
// Формат: {month feb ...}
char ch = 0;
if (is >> ch && ch! = '{') {
      is.unget ();
      is. clear (ios_base :: failbit); // Помилка введення Month
return is;
}

string month_marker;
string mm;
is >> month marker >> mm;
if (! is || month_marker! = "month")
        error ( "Наверное початок Month");
```

```
m.month = month_to_int (mm);

Reading r;
int duplicates = 0;
int invalids = 0;
for (Reading r; is >> r;) {
    if (is_valid (r)) {
        if (m.day [r.day] .hour [r.hour]! = not_a_reading)
++ duplicates;
m.day [r.day] .hour [r.hour] = r. temperature;
}
else
        ++ invalids;
}
if (invalids)
        error ( "Неверние дані в Month, всього", invalids);
if (duplicates)
        error ( "Повторюється свідчення в Month, всього", duplicates);
end_of_loop (is, '}', "Неправильний кінець Month");
return is;
}
```

Пізніше ми ще повернемося до функції month_to_int (): вона перетворюючи ет символічні позначення місяців, такі як jun, в число з діапазону на [0, 11]. Зверніть увагу на використання функції end_of_loop () для перевірки ознаки завершення введення. Ми підраховуємо кількість неправильних і повторюваних об'єктів класу Readings (ця інформація може комусь знадобитися).

Оператор >> в класі Month виконує грубу перевірку коректності об'єкта класу Reading, перш ніж записати його в пам'ять.

```
constexpr int implausible_min = -200;
constexpr int implausibe_max = 200;

bool is_valid (const Reading & r)
// Груба перевірка
{
        if (r.day <1 || 31 <r.day) return false;
        if (r.hour <0 || 23 <r.hour) return false;
        if (r.temperature <implausibe_min ||
        ішplausible_max <r. temperature)
        return false;
return true;
```

```
}
```

Нарешті, ми можемо прочитати об'єкти класу Year. Оператор >> в класі Year аналогічний оператору >> в класі Month.

```
istream & operator >> (istream & is, Year & y)
// Прочитуємо об'єкт класу Year з потоку is в об'єкт y
// Формат: {year один тисяча дев'ятсот сімдесят два}
{
char ch;
is >> ch;
if (ch! = '{')
is.unget ();
is.clear (ios :: failbit);
return is;
}

string year_marker;
int yy;
is >> year_marker >> yy;
if (! is || year_marker! = "year")
error ( "Неправильне початок Year");
y. year = yy;
while (true) {
        Month m;    // Щоразу створюємо новий об'єкт   m
        if (! (is >> m)) break;
        y. month [m. month] = m;
}
end_of_loop (is, '}', "Неправильний кінець Year");
return is;
}
```

Можна було б навіть сказати не просто "аналогічно", а "гнітюче аналогічно", якби не один важливий нюанс. Подивіться на цикл читання. Напевно, ви очікували чогось на кшталт

```
for (Month m; is >> m;)
      y.month [m.month] = m;
```

Можливо, так, оскільки саме так ми до сих пір записували все цикли введення. Саме цей фрагмент ми написали спочатку, і він виявився неправильним. Проблема в тому, що функція operator >> (is tream & is, Month &

m) присвоює об'єкту m абсолютно нового значення; вона просто додає в нього дані з об'єкта класу Reading. Таким чином, повторюється інструкція is >> m додавала б дані в один і той же об'єкт m. На жаль, в цьому випадку кожен новий об'єкт класу Month містив би всі свідчення всіх попередніх місяців поточного року. Для того щоб зчитувати дані з допомогою інструкції is >> m, нам потрібен зовсім новий, порожній об'єкт класу Month. Найпростіше помістити визначення об'єкта m в цикл так, щоб він на кожній ітерації створювався і ініціалізувати заново. В якості альтернативи можна було б зробити так, щоб функція operator >> (is tream & is, Month & m) перед зчитуванням в циклі надавала б об'єкту m порожній об'єкт.

```
for (Month m; is >> m;) {
      y. month [m. month] = m;
      m = Month {}; // "Реініціалізація" m
```

Спробуємо скористатися написаним.

```
// Відкриваємо файл введення
cout << "Введіть ім'я вхідного файлу \ n";
string iname;
cin >> iname;
ifstream ist {iname};
if (! ifs) error ( "Неможливо відкрити вхідний файл", iname);

// Генерація виключення в разі стану bad ()
ifs.exceptions (ifs.exceptions () ios_base :: badbit);

// Відкриваємо файл виводу
cout << "Введіть ім'я виходного файлу \ n";
string oname;
cin >> oname;
ofstream ost {oname};
if (! ofs) error ( "Неможливо відкрити вихідний файл '', oname);

// Читання довільної кількості років:
vector <Year> ys;
while (true) {
Year y; // На кожній ітерації отримуємо знову
          // ініціалізований об'єкт типу Year
if (! (ifs >> y)) break;
ys.push_back (y);
}
```

```
cout << "вважає" << ys.size () << "річних записів \ n";

for (Year & y: ys) print_year (ofs, y);
```

Лекція 15. Чисельна бібліотека мови С.

Для деяких категорій людей (наприклад, вчених, інженерів або статистиків) основним заняттям є серйозні числові розрахунки. У роботі багатьох людей такі розрахунки відіграють значну роль. До цієї категорії відносяться, наприклад, фахівці з комп'ютерних наук, іноді, що працюють з фізиками. У більшості людей необхідність в числових розрахунках, що виходить за рамки простих арифметичних дій над цілими числами і числами з плаваючою точкою, виникає досить рідко. Мета даної лекції - описати можливості мови С ++, що необхідні для вирішення простих обчислювальних задач.

15.1. Розмір, точність і переповнення

Коли ви використовуєте вбудовані типи і звичайні методи обчислень, числа зберігаються в областях пам'яті фіксованого розміру: інакше кажучи, цілочисельні типи (int, long і ін.) являють собою лише наближення математичного поняття цілих чисел, а числа з плаваючою точкою (float, double і ін.) є лише наближенням математичного поняття дійсних чисел. Звідси випливає, що з математичної точки зору деякі обчислення виявляються неточними або неправильними. Розглянемо приклад.

```
float x = 1. 0/333;
float sum = 0;
for (int i = 0; i <333; ++ i) sum + = x;
cout << setprecision (15) << sum << "\ n";
```

Виконавши цю програму, ми отримаємо не одиницю, а

```
0. 999999463558197
```

Ми очікували чогось подібного. Число з плаваючою точкою складається тільки з фіксованої кількості бітів, тому ми завжди можемо "зіпсувати" його, виконавши обчислення, результат якого складається з більшої кількості бітів, ніж допускає апаратне забезпечення. Наприклад, раціональне число 1/3 неможливо уявити точно як десяткове число (проте можна використовувати багато цифр його десяткового подання). Точно так само неможливо точно уявити число 1 / 333, тому, коли ми складаємо 333 копії числа x (найкраще машинне наближення числа 1 / 333 за допомогою типу float), то отримуємо число, трохи відрізняється від одиниці. При інтенсивному використанні чисел з плаваючою точкою виникає помилка округлення: залишається лише оцінити, наскільки сильно вона впливає на результат.

Завжди перевіряйте, наскільки точними є результати. При обчисленнях ви повинні уявляти собі, яким має бути результат, інакше легко будете обмануті помилкою обчислень або якоїсь дурної помилкою. Пам'ятайте про похибки

округлення і, якщо сумніваєтеся, зверніться за порадою до експерта або почитайте підручники з чисельних методів.

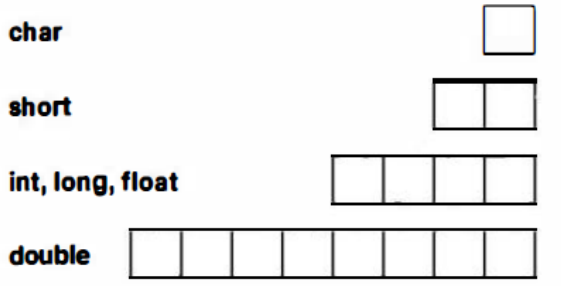
Вплив фіксованого розміру цілих чисел може проявитися більш вражаюче. Справа в тому, що числа з плаваючою точкою за визначенням є наближеннями дійсних чисел, тому вони можуть втрачати точність (тобто втрачати наймолодші значущі біти). Цілі ж числа часто переповнюються (тобто втрачають найстарші значущі біти). В результаті помилки. пов'язані з числами з плаваючою точкою, мають більш складний характер (які новачки часто не помічають). А помилки, що пов'язані з цілими числами, кидаються в очі (їх важко не помітити навіть новачкові). Ми вважаємо за краще, щоб помилки виявлялися якомога раніше, тоді їх легше виправити.

Розглянемо цілочисельну задачу.

```
short int y = 40000;
int i = 1000000;
cout << y << "" << i * i << "\ n";
```

Виконавши цю програму, отримаємо наступний результат:
-25536 -727 379 968

Цього і слід було очікувати. Тут ми бачимо ефект переповнення. Цілочисельні типи дозволяють представити лише відносно невеликі цілі числа. Нам просто не вистачить бітів, щоб точно представити кожне необхідне нам ціле число способом, що дозволяє виконувати ефективні обчислення. В даному випадку двухбайтове ціле типу short не може уявити значення 40 000, а чотирьохбайтове ціле типу int не може уявити число 1 000 000 000 000. Точні розміри вбудованих типів в мові C ++ залежать від апаратного забезпечення і компілятора; розмір змінної x або типу x в байтах можна визначити за допомогою оператора sizeof (x). За визначенням sizeof (char) = 1. Це можна проілюструвати наступним чином.



Це розміри для операційної системи Windows і компілятора компанії Microsoft. У мові C ++ є багато способів представити цілі числа і числа з плаваючою точкою, використовуючи різні розміри. але якщо у вас немає вагомих причин надходити інакше, краще дотримуватися типів char, int і double. У

більшості програм (але, зрозуміло, не у всіх) інші цілочисельні типи і типи з плаваючою точкою викликають більше проблем, ніж дають переваг.

Ціле число можна присвоїти змінній, що має тип з плаваючою точкою. Якщо ціле число виявиться більше, ніж може уявити даний тип числа з плаваючою точкою, відбудеться втрата точності. Розглянемо приклад.

```
cout << "Розміри:" << sizeof (int)
                << '' << sizeof (float) << '\ n';
int x = 2100000009; // Велике ціле число
float f = x;

cout << x << '         '<< f << endl;
cout << setprecision (15) << x << '         '<< f <<' \ n ';
```

На нашому комп'ютері ми отримали такий результат:

```
Розміри 4 4
2100000009              2.1e + 009
2100000009              2100000000
```

Типи float і int займають однакову кількість пам'яті (4 байта). Тип float складається з мантиси (як правило, значення між нулем і одиницею) і показника ступеня (тобто число має вигляд мантиса-$10^{\text{показатель ступеня}}$), тому він не може точно висловити найбільше число int. (Якби ми спробували зробити це, то звідки б ми взяли достатню кількість пам'яті для мантиси після розміщення показника ступеня?) Як і слід було очікувати, змінна f представляє число 2100000009 настільки точно, наскільки можливо. Однак остання цифра 9 занадто велика для точного представлення - саме тому ми і вибрали для ілюстрації це число.

З іншого боку, коли ми присвоюємо число з плаваючою точкою змінній цілочисельного типу, відбувається усічення: інакше кажучи, дрібна частина - цифри після десяткового дробу - просто відкидається. Розглянемо приклад.

```
float f = 2.8;
int x = f;
cout << x << '' << f << '\ n';
```

Значення змінної x виявляється рівним 2. Воно не буде дорівнює 3, як ви могли подумати, якщо застосували "правило округлення 4/5". У мові C ++ перетворення типу float в тип int виконується за допомогою усічення, а не округлення.

При обчисленнях слід побоюватися можливого переповнення і зрізання. Мова C ++ не вирішить цю проблему замість вас. Розглянемо приклад.

```
void f (int i, double fpd)
{
char з = i;           // Тип char дійсно представляє
                      // дуже маленькі цілі числа
short s = i;          // Небезпечно: тип int може не поміститися
                      // в пам'яті для змінної типу short
i = i + 1;            // Що буде, якщо число і до збільшення
                      // було максимально можливим?
long lg = i * i;      // Небезпечно: long може бути не більше int
float fps = fpd;      // Небезпечно: значення типу double може
                      // не поміститися в типі float
i = fpd;              // Усікання; наприклад, 5.7 -> 5
fps = i;              // Можлива втрата точності (при дуже
                      // великих цілочисельних значеннях)
}
void g ()
{
char ch = 0;
for (int i = 0; i <500; ++ i)
     cout << int (ch ++) << '\ t';
}
```

Якщо сумніваєтеся, поекспериментуйте! Не слід впадати у відчай і не слід обмежуватися читанням документації. Без експериментів ви можете не зрозуміти зміст вельми складної документації, що стосується числових типів.

Ми намагаємося обмежуватися деякими типами даних. Це дозволяє мінімізувати ймовірність помилок. Наприклад, використовуючи тільки тип double і уникаючи типу float, ми мінімізуємо ймовірність виникнення проблем, пов'язаних з перетворенням double у float. Фактично ми вважаємо за краще використовувати для обчислень тільки типи int, double і complex, для символів - тип char і для логічних значень - тип bool. Решта арифметичних типів ми використовуємо тільки при гострій потребі.

15.2. Межі числових діапазонів

Кожна реалізація мови C++ визначає властивості вбудованих типів в заголовних файлах <limits>, <climits>, <limits.h> і <float.h>, щоб програмісти могли перевірити межі діапазонів, встановити обмежувачі і т.д. Вони критично важливі при створенні низькорівневих інструментів. Якщо вас цікавлять ці значення, значить, ви працюєте безпосередньо з апаратним забезпеченням, хоча існують і інші застосування цієї інформації. Наприклад, досить часто виникають питання про тонкощі реалізації мови, наприклад "Наскільки великим є тип int?" або "Чи має знак тип char?" Знайти певні і правильні відповіді в системній документації

буває важко. а в стандарті вказані тільки мінімальні вимоги. Однак можна легко написати програму, яка знаходить відповіді на ці питання.

```
cout << "Кількість байтів в типі int:"
      << sizeof (int) << '\ n';
cout << "Найбільше число типу int:"
       << INT_MAX << endl;
cout << "Найменше число типу int:"
       << numeric limits <int> :: min () << '\ n';

if (numeric_limits <char> :: is_signed)
       cout << "Тип char має знак \ n";
else
       cout << "Тип char не має знак \ n";

// Найменша додатне значення:
char ch = numeric_limits <char> :: min ();
cout << "Найменше додатне значення char:"
       << ch << '\ n';
cout << "Значення int для char з найменшим"
       << "додатним значенням:" << int (ch) << '\ n';
```

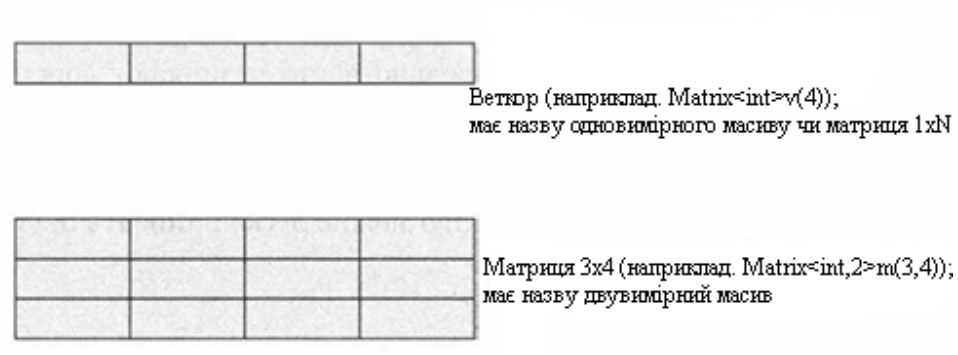
Якщо ви пишете програму, яка повинна працювати на різному апаратному забезпеченні, іноді виникає необхідність зробити цю інформацію доступною для вашої програми. Інакше вам доведеться "прошити" відповіді в програмі, ускладнивши її супровід.

Ці межі можуть бути також корисними для виявлення переповнення.

15.3. Масиви

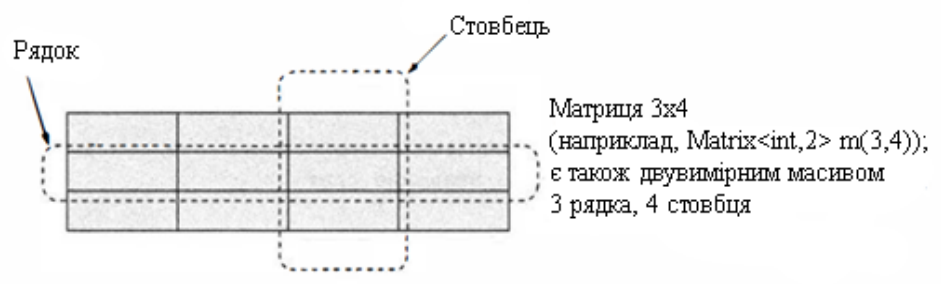
Масив (array) - це послідовність елементів, в якій доступ до кожного елементу здійснюється за допомогою його індексу (позиції). Синонімом цього поняття є вектор (vector). У цьому розділі ми приділимо увагу багатовимірним масивам, елементами яких також є масиви. Зазвичай багатовимірний масив називають матрицею (matrix). Різноманітність синонімів свідчить про популярність і корисності цього загального поняття. Стандартні класи vector, array, а також вбудований масив є одновимірними. А що якщо нам потрібен двовимірний масив (наприклад, матриця)? А якщо нам потрібні сім вимірювань?

Проілюструвати одно- і двовимірні масиви можна так.



Масиви мають фундаментальне значення в більшості обчислювач них завдань. Найбільш цікаві наукові, технічні, статистичні та фінансові обчислення тісно пов'язані з масивами.

Часто, кажучи про масиві, його розглядають як що складається з рядків і стовпців.



Стовпець - це послідовність елементів, що мають однакові перші координати (x-координати). Рядок - це безліч елементів, що мають однакові другі координати (y-координати).

15.4. Багатовимірні масиви в стилі мови C

Вбудований масив C ++ можна використовувати як багатовимірний масив. В цьому випадку багатовимірний масив інтерпретується як масив масивів, тобто масив, елементами якого є масиви. Розглянемо приклади.

```
int ai [4];                    // Одновимірна масив
double ad [3] [4];             // Двовимірний масив
char ac [3] [4] [5];  // Тривимірний масив
ai [1] = 7;
ad [2] [3] = 7.2;
ac [2] [3] [4] = 'з';
```

Цей підхід успадковує всі переваги і недоліки одновимірного масиву.

Переваги:

- Безпосереднє відображення на апаратне забезпечення.
- Ефективний для низькорівневих операцій.

- Безпосередня мовна підтримка.

Проблеми:

- Багатовимірні C-масиви є масивами масивів.
- Фіксовані на етапі компіляції розміри. Якщо ви хочете визначати розмір масиву на етапі виконання програми, використовуйте динамічну пам'ять.
- Масиви неможливо акуратно передати в функцію. При найменшій можливості масив перетворюється в покажчик на свій перший елемент.
- Ні перевірки діапазону. Як завжди, масив не знає свого розміру.
- Ні операцій над масивами, навіть присвоювання (копіювання).

Вбудовані масиви широко використовуються в числових розрахунках. Вони також є основним джерелом помилок і складнощів. Створення та налагодження таких програм у більшості людей викликають головний біль. На жаль, мова C ++ використовує багатовимірні масиви спільно з мовою C, так що є маса "зовнішніх" вихідних текстів, які їх використовують.

Основна проблема полягає в неможливості акуратною передачі багатовимірного масиву в функцію, так що доводиться працювати з покажчиками і виконувати явні обчислення, пов'язані з визначенням позицій в багатовимірному масиві. Розглянемо приклад.

```
void fl (int a [3] [5]);                // Тільки для матриць [3] [5]

void f2 (int [] [5], int dim1);         // Перша розмірність може бути змінної

void f3 (int [5] [], int dim2); // Помилка: друга розмірність
                                        // не може бути змінної

void f4 (int [] [], int dim1, int dim2);    // Помилка (Не буде працювати в будь-
                                            //якому випадку)
void f5 (int * m, int dim1, int dim2)           // Як не дивно, але працює
{
for (int i = 0; i <dim1; ++ i)
for (int j = 0; j <dim2; ++ j) m [i * dim2 + j] = 0;
}
```

Тут ми передаємо масив m як покажчик int *, незважаючи на те що насправді це двовимірний масив. Оскільки друга розмірність повинна бути змінною (параметром), у нас немає ніякої можливості повідомити компілятору, що масив m є масивом з розмірностями (dim1, dim2), так що ми просто передаємо покажчик на початок пам'яті, в якій він зберігається. Вираз m [i * dim2 + j] насправді означає m [i, j], але, оскільки компілятор не знає, що m є двовимірним масивом, ми повинні самі обчислювати позицію елемента m [i, j] в пам'яті.

Цей спосіб дуже складний, примітивний і вразливий для помилок. Він також занадто повільний, оскільки явне обчислення позиції елемента ускладнює

оптимізацію. Замість того щоб вчити вас, як впоратися з цією ситуацією, ми сконцентруємося на бібліотеці C ++, яка взагалі усуває проблеми, пов'язані з вбудованими масивами.

15.5. Бібліотека Matrix

Що ми взагалі хочемо від масиву / матриці в численних розрахунках?

- "Мій код повинен виглядати дуже схожим на опис масивів, викладене в більшості підручників з математики ".
- Це відноситься також до векторів, матриць і тензор.
- Перевірка на етапах компіляції і виконання програми.
- Масиви будь-якої розмірності.
- Масиви з будь-якою кількістю елементів в будь-якої розмірності.
- Масиви є повноцінними змінними / об'єктами.
- Їх можна передавати куди завгодно.
- Звичайні операції над масивами.
- Індексування: ().
- Зрізання: [].
- Присвоєння: =.
- Операції масштабування (+ =, - =, * =.% = і т.д.).
- Змішані векторні операції (наприклад, res [i] = a [i] * c +b [2]).
- Скалярний добуток (res = сума a [i] *b [i]: відомо також як inner_product).
- Повинно забезпечуватися перетворення традиційних Позначений ний, пов'язаних з масивами / векторами, в код, який інакше ви повинні були б з великими витратами праці писати самостійно (і забезпечувати як мінімум таку ж ефективність).
- Масиви при необхідності можна збільшувати (при їх реалізації не використовуються "магічні" числа).

Бібліотека Matrix робить це і тільки це. Якщо ви хочете більшого, то повинні самостійно написати складні функції обробки масивів, розріджених масивів, управління розподілом пам'яті і так далі або використовувати іншу бібліотеку, яка краще відповідає вашим потребам. Однак багато хто з цих потреб можна задовольнити за допомогою алгоритмів і структур даних, надбудованих над бібліотекою Matrix. Бібліотека Matrix не є частиною стандарту ISO C ++. Її можливості визначені в просторі імен Numeric_lib. Ми вибрали назву Matrix, тому що слова "вектор" і "масив" і без того надмірно використовуються в бібліотеках мови C ++. Реалізація бібліотеки Matrix використовує складні методи. які тут не описуються.

Розмірності і доступ.

Розглянемо простий приклад.

```
#include "Matrix.h"
using namespace Numeric_lib;
```

```
void f (int n1, int n2, int n3)
{
Matrix <double, 1> ad1 (n1);    // Одновимірна з елементами double
Matrix <int, 1> ai1 (n1);       // Одновимірна з елементами int
ad1 (7) = 0;                    // Індексування в стилі Fortran - ()
adl [7] = 8;                    // Індексування в стилі C - []

Matrix <double, 2> ad2 (n1, n2);       // Двовимірна
Matrix <double, 3> ad3 (nl, n2, n3);   // Тривимірна
ad2 (3, 4) = 7.5;                      // Істинне багатовимірне
ad3 (3,4, 5) = 9.2;                    // індексування
```

Таким чином, визначаючи матрицю (об'єкт класу Matrix), ви повинні вказати тип елемента і кількість розмірностей. Очевидно, що клас Matrix є шаблонною, а тип елементів і кількість розмірностей є шаблонні параметри. Результатом передачі пари шаблонних параметрів класу Matrix (наприклад, Matrix <double, 2>) є тип (клас), за допомогою якого можна визначити об'єкти, вказавши аргументи (наприклад, Matrix <double, 2> ad2 (n1, n2)), які задають розмірності. Так, змінна ad2 є двовимірним масивом з розмірностями n1 і n2, яку також називають матрицею n1 x n2. Для того щоб отримати елемент оголошеного типу з одновимірного об'єкта класу Matrix, слід вказати один індекс. Для того щоб отримати елемент оголошеного типу з двовимірного об'єкта класу Matrix, слід вказати два індекси, і т.д.

Як і у вбудованих масивах і векторах vector, елементи в об'єкті класу Matrix індексуються з нуля; інакше кажучи, елементи Matrix нумеруються в діапазоні [0, max), де max - кількість елементів.

Це просто і "взято прямо з підручника". Якщо у вас виникнуть проблеми, потрібно лише звернутися до відповідного підручника з математики, а не до керівництва з програмування. Єдина тонкість тут полягає в тому, що можна не вказувати кількість розмірностей в Matrix: за замовчуванням матриця є одновимірної. Зверніть увагу також на те, що ми можемо використовувати індексування як за допомогою оператора [] (в стилі мов C і C ++), так і за допомогою оператора ().

Це дозволяє нам краще справлятися з великою кількістю розмірностей. Індекс [x] завжди означає єдиний індекс, виділяючи окремий рядок в об'єкті класу Matrix; якщо змінна a є n-мірної матрицею Matrix, то a [x] - це n-1 -мірним матриця. Позначення ж (x, y, z) має на увазі одночасне використання декількох індекс інфляції сов, виділяючи відповідний елемент об'єкта класу Matrix; кількість індексів повинна дорівнювати кількості розмірностей.

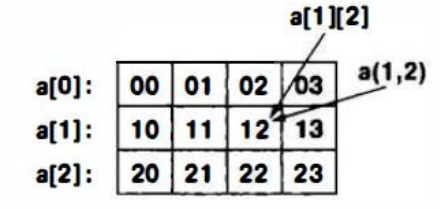
Подивимося, що станеться, якщо ми зробимо помилку.

```
void f (int n1, int n2, int n3)
```

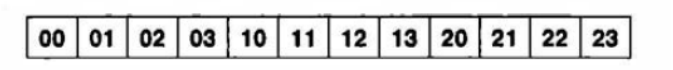
```
{
Matrix <int, 0> ai0;                    // Помилка: 0-вимірна матриця
Matrix <double, 1> ad1 (5);
Matrix <int, 1> ai (5);

Matrix <double, 1> ad11 (7);
ad1 (7) = 0;                    // Виняток Matrix error
                                // (7 - за межами діапазону)
ad1 = ai;               // Помилка: різні типи елементів
ad1 = ad11;             // Виняток Matrix_error
                        // (незбіжні розмірності)

Matrix <double, 2> ad2 (n1); // Помилка: не вказана друга розмірність
ad2 (3) = 7.5;                  // Помилка: неправильна кількість індексів
ad2 (1, 2, 3) = 7.5;            // Помилка: неправильна кількість індексів

Matrix <double, 3> ad3 (n1, n2, n3);
Matrix <double, 3> ad33 (n1, n2, n3);
ad3 = ad33;             // ОК: однакові типи елементів,
                        // одинакові розмірності
}
```

Невідповідності між оголошеним кількістю розмірностей і їх використанням виявляють на етапі компіляції. Вихід за межі діапазону перехоплюється на етапі виконання програми; при цьому генерується виключення Matrix_error.

Перша розмірність матриці - це рядок, а друга - стовпець, тому ми індексуємо двовимірну матрицю (двовимірний масив) як (рядок, стовпець). Можна також використовувати позначення [рядок] [стовпець], так як індексування двовимірної матриці за допомогою одновимірного індексу породжує одновимірну матрицю - рядок. Цю ситуацію можна проілюструвати наступним чином.



Елементи матриці розміщуються в пам'яті через підрядник.



Оскільки клас Matrix знає свою розмірність, його елементи можна передавати як аргумент без проблем:

```
void init (Matrix <int, 2> & a)    // Ініціалізація кожного елемента
{                                  // характеризующім значенням
   for (int i = 0; i <a.dim1 (); ++ i)
   for (int j = 0; j <a.dim2 ();      ++ j)
   a (i, j) = 10 * i + j;
}
void print (const Matrix <int, 2> & a) // Порядковий висновок матриці
{
for (int i = 0; i <a.dim1 (); ++ i)
   for (int j = 0; j <a.dim2 (); ++ j)
        cout << a (i, j) << '\ t';
     cout << '\ n';
   }
}
```

Виклик dim1 () повертає кількість елементів в першій розмірності, dim2 () - кількість елементів у другій розмірності і т.д. тип елементів і кількість розмірностей є частиною типу Matrix, тому неможливо написати функцію, яка одержує довільну матрицю Matrix як аргумент (але можна написати відповідний шаблон).

```
void init (Matrix & a);      // Помилка: пропущені тип елементів
                             // і кількість розмірностей
```

Зверніть увагу на те, що бібліотека Matrix не містить матричних операцій. таких як складання двох чотиривимірних матриць або множення двовимірних матриць на одномірні. Елегантна і ефективна реалізація цих операцій виходить за рамки даної бібліотеки. Відповідні матричні бібліотеки можна надбудувати над бібліотекою Matrix.

Одновимірна матриця.

Що можна зробити з найпростішої матрицею - одновимірної?

Кількість розмірностей в оголошенні такого об'єкта можна не вказувати, тому що за замовчуванням воно дорівнює одиниці.

```
Matrix <int, 1> a1 (8); // a1 - одномірна матриця цілих чисел
Matrix <int> a (8); // Чи означає Matrix <int, 1> a (8);
```

Таким чином, об'єкти a й a1 мають однаковий тип (Matrix <int, 1>). У кожного об'єкта класу Matrix можна запросити загальна кількість елементів і

кількість елементів в певному вимірі. У одновимірної матриці ці параметри збігаються.

```
a. size ();      // Количеств про елементів в Matrix
a. dim1 ();    // Количеств про елементів в першому вимірі
```

Можна також звертатися до елементів матриці. використовуючи схему їх розташування в пам'яті, тобто через покажчик на перший елемент.

```
int * p = a. data (); // Отримуємо дані через покажчик на масив
```

Це корисно при передачі об'єктів класу Matrix функцій в стилі мови C, які приймають в якості аргументів покажчики. Матриці можна індексувати.

```
a (i);          // i -й елемент (в стилі Fortran) з перевіркою діапазону
a [i];          // i -й елемент (в стилі C) з перевіркою діапазону
a (1,2);        // Помилка: a - одновимірна матриця Matrix
```

Багато алгоритми звертаються до частини матриці Matrix. Така частина називається "зрізанням" (slice), являє собою підматрицю, або діапазон елементів, і створюється функцією slice (). У класі Matrix є два варіанти цієї функції.

```
a.slice (i);            // Елементи з a [i] по останній
a.slice (i, n);         // n елементів з a [i] по a [i + n-1]
```

Індекси і зрізання можна використовувати як в лівій частині оператора присвоювання, так і в правій. Вони посилаються на елементи об'єкта класу Matrix, не створюючи їх копії. Розглянемо приклад.

```
a.slice (4,4) = a.slice (0,4); // Надаємо першу половину матриці другий
```

Наприклад. якщо об'єкт a спочатку виглядав як

```
{1 2 3 4 5 6 7 8}
```

то в результаті ми отримаємо

```
{1 2 3 4 1 2 3 4}
```

Зверніть увагу на те, що найчастіше зрізання задаються початковим і кінцевим елементами Matrix; т. е. a.slice (0, j) - це діапазон [0, j). а a.slice (j) - діапазон [j, a.size ()). Зокрема, наведений вище приклад можна легко переписати:

```
a.slice (4) = a.slice (0,4); // Надаємо першу половину
```

// матриці другий

Інакше кажучи, позначення - як кому подобається.

Ви можете вказати такі індекси i і n, що a.slice (i, n) вийде за межі діапазону матриці a. Однак отримана зрізка буде мати тільки ті елементи, які дійсно належать об'єкту a. Наприклад, зрізка a.slice (i, a.size ()) означає діапазон [i, a.size ()), а a.slice (a.size ()) і a.slice (a.size (), 2) - порожні об'єкти класу Matrix. Це виявляється корисним у багатьох алгоритмах. Ми запозичили таку угоду з математики. Очевидно, що зрізає a.slice (i, 0) є порожнім об'єктом класу Matrix. Ми б не писали це навмисно, але існують алгоритми. які стають простіше, якщо зрізка a.slice (i, n) при параметрі n, рівному 0, є марною матрицею (а не призводить до помилки, якої потрібно було б уникати).

При звичайному копіюванні (об'єктів C ++) виконується копіювання всіх елементів.

```
Matrix <int> 2 = a; // копіює ініціалізацію
a = A2;             // копіює присвоювання
```

Можна застосовувати вбудовані операції до кожного елементу Matrix:

```
a * = 7;            // a [i] * = 7 для всіх i (працює також
                    // для + =, - =, / = і т. д.)
a = 7;              // a [i] = 7 для всіх i
```

Це відноситься до кожного оператору присвоювання і кожному складеному оператору присвоювання (=, + =, - =, / =, * =,% =, "=, & =, | =, >> =, << =) за умови, що тип елемента підтримує відповідний оператор. Крім того, можна застосовувати функції до кожного елементу Matrix:

```
a.apply (f);        // a [i] = f (a [i]) для кожного елемента a [i]
a.apply (f, 7);     // a [i] = f (a [i], 7) для кожного елемента a [i]
```

Складові оператори присвоювання і функція apply () модифікують елементи своїх аргументів типу Matrix. Якщо ж ми захочемо створити нову матрицю, то можемо використовувати інший виклик:

```
b = apply (abs, a); // Створюємо нову матрицю, таку, що b (i) == abs (a (i))
```

Функція ABS - це функція обчислення абсолютної величини з стан дротяні бібліотеки. По суті виклик apply (f, x) пов'язаний з викликом x.apply (f) точно так же, як оператор + пов'язаний з оператором + =. Розглянемо приклад.

```
b = a * 7;          // b [i] = a [i] * 7 для кожного i
```

```
a * = 7;              // a [i] = a [i] * 7 для кожного i
y = apply (f, x);     // y [i] = f (x [i]) для кожного i
x.apply (f);          // x [i] = f (x [i]) для кожного i
```

В результаті ми отримаємо a == b і x == y.

Крім того, для того щоб забезпечити відповідність з варіантом функції-члена apply з двома аргументами (a.apply (f, x)), ми надали

```
b = apply (f, a, x);   // b [i] = F (a [i], x) для кожного i
```

наприклад:

```
double scale (double d, double s) {return d * s; }
b = apply (scale, a, 7);      // b [i] = a [i] * 7 для кожного i
```

Зверніть увагу на те, що "автономна" функція apply () приймає в якості аргументу функцію, що вертає результат, обчислюючи його з переданих їй аргументів, а потім використовує цей результат для ініціалізації результуючого об'єкта типу Matrix. Як правило, це не призводить до зміни матриці. до якої користуєтеся цією функцією. Функція ж член apply () відрізняється тим, що приймає в якості аргументу функцію, модифікуючи її аргументи; т. е. ця функція модифікує елементи матриці. до яких застосовується. Розглянемо приклад.

```
void scale_in_place (double & d, double s) {d * = s;}
b.apply (scale_in_place, 7);              // b [i} * = 7 для кожного i
```

У класі Matrix передбачені також найбільш корисні функції з традиційних математичних бібліотек:

```
Matrix <int> a3 =
        scale_and_add (a, 8, a2); // Злітне множення і додавання
int r = dot_product (a3, a);      // Скалярний добуток
```

Операцію scale_and_add () часто називають злитим множенням і складанням (fused multiply-add), або просто fma; її визначення виглядає так: result (i) = arg1 (i) * arg2 + arg3 (i) для кожного і в матриці. Скалярний добуток. відоме також під ім'ям inner_product; його визначення виглядає так: result + = arg1 (i) * arg2 (i) для кожного і в матриці, де початкове значення result звичайно дорівнює нулю.

Одномірні масиви дуже широко поширені; їх можна перед ставити як у вигляді вбудованого масиву, так і за допомогою класів vector і Matrix. Клас Matrix слід застосовувати тоді, коли необхідно виконувати матричні операції,

такі як * =, або коли одномірна матриця повинна взаємодіяти з іншими матрицями з більш високою розмірністю.

Корисність цієї бібліотеки можна пояснити тим, що вона "краще узгоджена з математичними операціями", а також тим, що при її використанні для роботи з кожним елементом матриці не доводиться писати цикли. У будь-якому випадку в результаті ми отримуємо більш короткий код і менше можливостей зробити помилку. Операції класу Matrix, такі як копіювання, присвоювання всіх елементів і операції над усіма елементами, дозволяють не використовувати цикли (а значить, можна не турбуватися про пов'язаних з ними проблеми).

Клас Matrix має два конструктора для копіювання даних з вбудованих масивів в матриці. Розглянемо приклад.

```
void some_function (double * p, int n)
{
double val [] = {1.2, 2.3, 3.4, 4.5};
Matrix <double> data {p, n};
Matrix <double> constants {val};
//. . .
}
```

Це часто буває корисним, коли ми отримуємо дані в вигляді звичайних масивів або векторів, створених в інших частинах програми, які не використовують об'єкти класу Matrix.

Зверніть увагу на те, що компілятор може самостійно визначити кількість елементів в ініціалізованому масиві, тому це число при визначенні об'єкта constants вказувати необов'язково - воно дорівнює 4. З іншого боку, якщо елементи задані всього лише покажчиком, то компілятор не знає їх кількості, тому при визначенні об'єкта data ми повинні задати як покажчик p, так і кількість елементів n.

Двовимірні матриці.

Загальна ідея бібліотеки Matrix полягає в тому, що матриці різної розмірності насправді в більшості випадків дуже схожі, за винятком ситуацій, в яких особливу роль відіграють розмірності. Таким чином, більшість з того, що ми можемо сказати про одновимірних об'єктах класу Matrix, відноситься і до двовимірним матрицями.

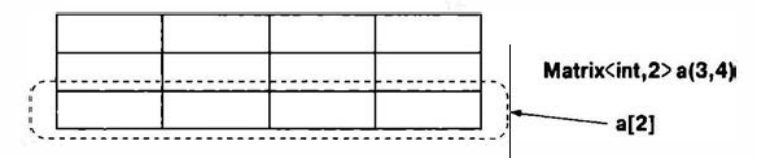
```
Matrix <int, 2> a (3,4);
int s = a.size ();          // Кількість елементів
int d1 = a.dim1 ();         // Кількість елементів в рядку
int d2 = a.dim2 ();         // Кількість елементів в стовпці
int * p = a. data ();       // Покажчик на дані
```

Ми можемо запросити загальна кількість елементів і кількість елементів в кожній розмірності. Крім того, можемо отримати покажчик на елементи матриці, розміщені в пам'яті.

Ми можемо використовувати індекси.

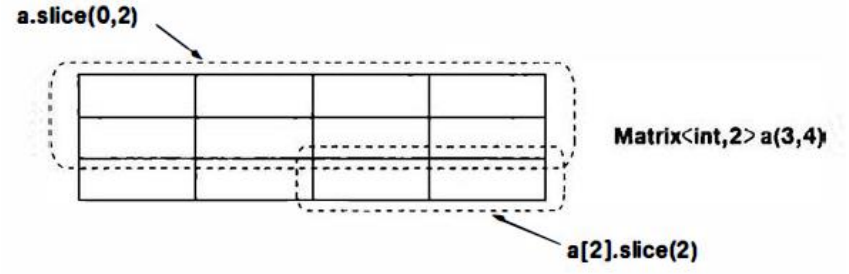
```
a (i, j);       // Елемент (i, j) (в стилі Fortran) з перевіркою діапазону
a [i];          // i -я рядок (в стилі C) з перевіркою діапазону
a [i] [j];      // Елемент (i, j) (в стилі C)
```

У двовимірної матриці індексування за допомогою конструкції [i] створює одновимірну матрицю, що представляє собою i-й рядок вихідної матриці. Це означає, що ми можемо витягувати рядки і передавати їх операторам і функцій, які вимагають в якості аргументів одномірні матриці і навіть вбудовані масиви (за допомогою a [i] .data ()). Зверніть увагу на те, що індексування виду a(i,j) може виявитися швидше, ніж індексування виду a[i][j], хоча це сильно залежить від компілятора і оптимізатора.



Ми можемо отримати зрізання.

```
a.slice (i); // Рядки від a [i] до останньої
a.slice (i, n); // Рядки від a [i] до a [i + n-1]
```



Зрізання двовимірної матриці сама є двовимірною матрицею (можливо, з меншою кількістю рядків).

Розподілені операції над двовимірними матрицями такі ж, як і над одновимірними. Цими операціями не важливо, як саме зберігаються елементи; вони просто застосовуються до всіх елементів в порядку їх слідування в пам'яті.

```
Matrix <int, 2> 2 = a;      // копіює ініціалізація
a = A2;                     // копіює присвоювання
a * = 7;                    // Масштабування (а також + =, - = і т.д.)
```

```
a.apply (f);                // a (i, j) = f (a (i, j)) для кожного a (i, j)
a.apply (f, 7);             // a (i, j) = f (a (i, j), 7) для кожного a (i, j)
b = apply (f, a);           // Нова матриця з b (i, j) == f (a (i, j))
b = apply (f, a, 7);        // Нова матриця з b (i, j) == f (a (i, j), 7)
```

Виявляється, що перестановка рядків також буває корисною, тому ми передбачили і її.

```
a. swap_rows (1,2);         // Перестановка рядків a [l] <-> a [2]
```

Перестановки стовпців swap_columns () немає. Через порядкової схеми зберігання матриць в пам'яті рядки і стовпці не зовсім рівноправні. Ця асиметрія проявляється також у тому, що оператор [i] повертає тільки рядок (для стовпців аналогічний опера тор не передбачений). У записі (i, j) перший індекс і вибирає рядок. Ця асиметрія має глибокі математичні коріння.

Кількість дій, які можна було б виконати над двовимірними матрицями, здається нескінченним.

```
enum Piece {none, pawn, knight, queen, king, bishop, rook};
Matrix <Piece, 2> board (8,8);          // Шахова дошка

const int white_start_row = 0;
const int back_start_row = 7;

Matrix <Piece> start row
        = {Rook, knight, bishop, queen, king, bishop, knight, rook};

        Matrix <Piece> clear_row (8); // 8 елементів зі значеннями за замовчуванням
```

Ініціалізація об'єкта clear_row використовує те, що none == 0, і те, що елементи за замовчуванням не започатковано нулем.

Ми можемо використовувати start_row і clear_row наступним чином:

```
board [white_start_row] = start_row; // Розстановка білих фігур
for (int i = 1; i <7; ++ i)
        board [i] = clear_row; // Очищення середини дошки
board [black_start_row] = start_row; // Розстановка чорних фігур
```

Зверніть увагу, що, коли ми витягли рядок, використовуючи вираз [i], ми отримали lvalue; інакше кажучи, ми можемо виконувати присвоювання результату операції board [i].

Бібліотека Matrix надає дуже прості засоби введення і виведення одно- і двовимірних матриць:

```
Matrix <double> a (4);
cin >> a;
cout << a;
```

Цей фрагмент коду читає чотири розділених пробілами числа типу double, укладених у фігурні дужки; наприклад:

```
{1.2 3.4 5.6 7.8}
```

Висновок аналогічний, тому ми просто побачимо те, що ввели.

Механізм введення-виведення двовимірних матриць просто зчитує і записує послідовності одновимірних матриць, укладені у фігурні дужки. Розглянемо приклад.

```
Matrix <int, 2> m (2, 2);
cin >> m;
cout << m;
```

Цей код читає запис

```
{
{1 2}
{3 4}
}
```

Висновок виявляється таким же.

Оператори << і >> з класу Matrix надані для того, щоб спростити написання простих програм. У більш складних ситуаціях вам доведеться замінити їх своїми операторами. З цієї причини визначення операторів << і >> класу Matrix поміщено в заголовки MatrixIO.h (а не в Matrix.h). Так що, для того, щоб використовувати матриці в своїй програмі, вам необов'язково включати заголовок MatrixIO.h.

Тривимірні матриці.

По суті, тривимірні матриці, як і матриці більш високих розмірностей, дуже схожі на двовимірні; просто вони мають більше розмірностей. Розглянемо приклад.

```
Matrix <int, 3> a (10, 20, 30);
a.size ();              // Кількість елементів
a.dim1 ();              // Кількість елементів в розмірності 1
a.dim2 ();              // Кількість елементів в розмірності 2
```

```
a.dim3 ();                  // Кількість елементів в розмірності 3
int * p = a. data (); // Покажчик на дані
a (i, j, k);                // (i, j, k) -й елемент (в стилі Fortran)
                            // з перевіркою діапазону
a [i];                      // i -я рядок (в стилі C) з перевіркою діапазону
a [i] [j] [k];              // (i, j, k) -й елемент (в стилі C)
a.slice (i);                // Рядки від i -й до останньої
a.slice (i, j);             // Рядки від i-ої до j -й
Matrix <int, 3> 2 = a       // копіює ініціалізація
a = A2;                     // копіює присвоювання
a * = 7;                    // Масштабування (також + =, - = і т. Д.)
a.apply (f);                // a (i, j, k) = f (a (i, j, k))
                            // для кожного елемента a (i, j, k)
a.apply (f, 7);                     // a (i, j, k) = f (a (i, j, k), 7)
                            // для кожного елемента a (i, j, k)
b = apply (f, a);           // Створення нової матриці, такий, що
                            // b (i, j, k) == f (a (i, j, k))
b = apply (f, a, 7);        // Створення нової матриці, такий, що
                            // b (i, j, k) == f (a (i, j, k), 7)
a. swap_rows (7,9);         // Перестановка рядків a[7] <-> a [9]
```

Якщо ви розумієте, як працювати з двовимірними матрицями, то зможете працювати і з тривимірними. Наприклад, тут a - тривимірна матриця, тому a [i] - двовимірна матриця (за умови, що індекс i не виходить за межі допустимого діапазону): a[i][j] - матриця одномірна (за умови, що індекс j НЕ виходить за межі допустимого діапазону); ну а a[i][j][k] - елемент типу int (за умови, що індекс k не виходить за межі допустимого діапазону).

Оскільки ми бачимо світ тривимірним, при моделюванні найчастіше використовуються тривимірні матриці (наприклад, у фізичному моделюванні в декартовій системі координат).

```
int qrid_nx;          // Дозвіл сітки; задається на початку
int qrid_ny;
int qrid_nz;
Matrix <double, 3> cube (grid_nx, qrid_ny, qrid_nz);
```

Додавши в якості четвертого виміру час, ми отримаємо чотиривимірний простір, для опису якого потрібні чотиривимірні матриці. І так далі.

15.6. Випадкові числа

В якості практичного інструменту і математичної задачі випадкові числа в даний час досягли високого ступеня інтелектуальності, відповідної їх важливості

в реальному світі. Тут ми торкнемося лише азів теорії випадкових чисел, необхідних для простого тестування і моделювання. В заголовки <random> стандартної бібліотеки є набір складних засобів для генерації випадкових чисел, що відповідають різним математичним розподілом. Ці кошти стандартної бібліотеки засновані на двох фундаментальних поняттях.

- ***Механізми.*** Механізм являє собою функціональний об'єкт. який генерує рівномірно розподілене послідовність цілочисельних значень.
- ***Розподіли.*** Розподіл є функціональний об'єкт. який відповідно до математичною формулою генерується послідовність значень для даної вхідний послідовності значень, що генерується механізмом.

Наприклад, розглянемо функцію random_vector (). Виклик функції random_vector (n) породжує матрицю Matrix <double, 1>, що містить n елементів типу double з випадковими значеннями в діапазоні [0, n):

```
Vector random_vector (Index n)
{
Vector v (n);

default_random_engine ran {};          // Генерує цілі числа
uniform real distribution <>
ureal {0, max};                        // Показує int в double
                                       // в діапазоні [0, max)
for (Index i = 0; i <n; ++ i)
    v (i) = ureal (ran);
return v;
}
```

Механізм за замовчуванням (defaul t_random_engine) простий, малозатратен при роботі і досить хороший для несистематичного застосування. Для більш професійного застосування стандартна бібліотека пропонує різні механізми з кращими властивостями випадковості і різною вартістю виконання. Прикладами є linear_congurential_engine, mersenne_twister_engine і random_device. Якщо ви хочете використовувати механізми стандартної бібліотеки і вам потрібно щось краще default_random_engine, вам доведеться звернутися до документації.

Два генератора випадкових чисел з std_lib_facilities.h визначені в такий спосіб:

```
int randint (int min, int max)
{
static default_random_enqine ran;
return uniform_int_distribution <> {min, max} (ran);
```

```
}
int randint (int max)
{
return randint (0, max);
}
```

Ці прості функції найбільш корисні, але просто щоб випробувати щось ще, давайте згенеруємо випадкові числа, що підкоряються нормальному розподілу:

```
auto gen = bind (normal_distribution <double> {15,4.0},
                 default_random_engine {});
```

Функція стандартної бібліотеки bind () з заголовки <functional> створює функціональний об'єкт. який при використанні викликає перший аргумент з другим аргументом в якості свого аргументу. Так що в даному випадку виклики gen () призводять до повернення значень з нормальним розподілом, із середнім значенням 15 і дисперсією 4.0 за допомогою механізму default_random_engine.

15.7. Стандартні математичні функції

У стандартній бібліотеці містяться не менш стандартні математичні функції (cos, sin, log і т.д.). Їх оголошення можна знайти в заголовки <cmath>.

Стандартні математичні функції:

ABS (x)	Абсолютне значення
ceil (x)	Найменша ціле число, яке задовольняє умові> = x
floor (x)	Найбільше ціле число, яке задовольняє умові <= x
sqrt (x)	Квадратний корінь; значення x повинно бути невід'ємним
cos (x)	косинус
sin (x)	синус
tan (x)	тангенс
acos (X)	арккосинус; результат ненегативний
asin (x)	арксинус; повертається результат, найближчий до нуля
atan (X)	арктангенс
sinh (x)	гіперболічний синус
cosh (x)	гіперболічний косинус
tanh (x)	гіперболічний тангенс
exp (x)	Експонента (підстава натуральних логарифмів e в ступені x)
log (x)	Натуральний логарифм (з підставою e); значення x повинно бути позитивним
log10 (x)	десятковий логарифм

Стандартні математичні функції можуть мати аргументи типів float, double, long double і complex. Ці функції дуже корисні при обчисленнях з

плаваючою крапкою. Більш детальна інформація міститься в широко доступною документації, а для початку можна звернутися до довідкової системи вашого компілятора.

Якщо стандартна математична функція не може дати математично коректний результат, вона встановлює прапорець errno. Розглянемо приклад.

```
errno = 0;
double s2 = sqrt (-1);
if (errno) cerr << "Щось десь пішло не так, як треба";
if (errno = = EDOM)        // Помилка через вихід аргументу
                           // за межі обла сті визначення
        cerr << "Фунція sqrt () для негативних"
             << "аргументів не визначена";
pow (very_large, 2);       // Погана ідея
if (errno == ERANGE)       // Помилка через вихід за межі допустимого
                           діапазону
cerr << "pow (" << very_large
        << ", 2) занадто велика кількість дпя double";
```

Виконуючи серйозні математичні обчислення. ви завжди повинні перевіряти значення errno, щоб переконатися, що після повернення результату воно як і раніше нульове. В іншому випадку щось пішло не так, як треба. Для того щоб дізнатися, які математичні функції можуть встановлювати прапорець errno і чому він може дорівнювати, зверніться до документації.

Як показано в прикладі. нульове значення прапорця errno просто означає. що щось пішло не так. Функції, які не входять в стандартну бібліотеку. також досить часто встановлюють прапорець errno при виявленні помилок, тому слід точніше аналізувати різні значення змінної errno, щоб зрозуміти, що саме сталося. Якщо ви перевіряєте значення errno відразу після виконання функції стандартної бібліотеки і встановили її рівною нулю перед викликом, то ви можете покладатися на значення цієї змінної, як ми зробили це в наведеному вище прикладі для значень EDOM і ERANGE. Константа EDOM означає помилку, що виникла через вихід аргументу за межі області визначення функції (domain error), тобто проблему з аргументами. Константа ERANGE означає вихід за межі допустимого діапазону значень (range error). тобто проблему з повертається значенням.

15.8. Узагальнені чисельні алгоритми

Для використання чисельних алгоритмів потрібно підключити заголовний файл <numeric>.

accumulate

Перша форма алгоритму accumulate використовується для накопичення суми елементів послідовності, заданої ітераторами first і last. Початкове значення суми (зазвичай це 0) задається третім параметром. Тип цього параметра визначає тип результату (функція повертає обчислену суму):

```
template <class In. class T>
        T accumulate (In first, In last, T init);
```

Друга форма алгоритму accumulate дозволяє виконувати над третім параметром і черговим елементом послідовності задану операцію:

```
template t <class In, class T, class BinOp>
        T accumulate (In first, In last, T init, BinOp binary_op);
```

У наведеному далі прикладі обчислюється сума, добуток і сума квадратів елементів масиву:

```
#include <1ostream>
#include <numeric>
#include <functional>
using namespace std;
int sumkv (int s, int x) {return s + x * x;}
int main () {
const int m = 5;
int a [m] = {3, 2, 5, 1, 6}, sum = 0, mul = 1, sum2 = 0;
cout << accumulate (a, a + m, sum) << endl;
cout << accumulate (a, a + m, mul, multipTies <int> ()) << endl;
cout << accumulate (a, a + m, sum2.sumkv) << endl;
return 0:
}
```

inner_product

Перша форма алгоритму inner_product використовується для обчислення скалярного добутку двох послідовностей (скалярним твором послідовностей a й b є вираз 2ai * bi). Початкове значення твору задається четвертим параметром. Тип цього параметра визначає тип результату (функція повертає обчислений добуток):

```
template t <class In1, class In2, class T>
T inner_product (In1 first1, In1 last1, In2 first2,
        T init);
```

Друга форма алгоритму inner_product використовується для виконання над двома послідовностями дій, заданих за допомогою двох функцій або функціональних об'єктів. Перший використовується замість операції додавання, другий - замість множення:

```
template t <class In1, class In2, class T,
        class BinOp1, class Bin0p2>
    T inner_roduct (In1 first1, In1 last1, In2 first2,
    T init, Bin0p1 binary_opl, Bin0p2 binary_op2);
```

Наступний виклик обчислює твір сум відповідних елементів послідовностей a й b:

```
cout << inner_product (a, a + m, b, mul, multiplies <int> (), plus <int> ());
```

adjacent_difference

Алгоритм adjacent_difference виконує обчислення різниці між суміжними елементами, тобто $d_i = a_i - a_{i-1}$. Замість різниці четвертим параметром можна задати іншу операцію. Функція повертає ітератор на елемент, наступний за кінцем результуючої послідовності.

```
template t <class In, class Out>
Out adjacent_difference (In first, In last, Out result);
template t <class In, class Out, class BinOp>
Out adjacent_difference (In first, In last, Out result, BinOp binary_op);
```

Цей алгоритм є зворотним попередньому, тобто виклик для однієї і тієї ж послідовності спочатку одного, а потім іншого алгоритму призведе до вихідної послідовності.

Клас valarray

Для ефективної роботи з масивами чисел в стандартній бібліотеці визначено шаблонний клас Valarray. Операції з цим класом реалізовані з розрахунком на їх підтримку в архітектурі високопродуктивних систем. У бібліотеці описані також чотири допоміжних класу, що дозволяють отримати різні підмножини valarray: slice_array, gslice_array, mask_array і indirect_array.

Всі ці класи реалізовані як шаблони класів з параметром, що представляє собою тип елемента масиву.

Для використання класу valarray і пов'язаних з ним коштів потрібно підключити до програми заголовки <valarray>. У ньому описані, крім перерахованих шаблонів класів, класи slice і gslice, що задають підмножини індексів масиву, а також заголовки функцій, призначених для роботи з цими шаблонами і класами.

Шаблон slice_array є рядком, стовпець або інше регулярне підмножина елементів valarray (наприклад, його парні елементи). За допомогою цього шаблону можна представляти valarray як матрицю довільної розмірності.

Шаблон gslice_array дозволяє задавати більш складні закони формування підмножини масиву, а mask_array - довільні підмножини за допомогою бітової маски. Шаблон indirect_array містить не самі елементи масиву, і їх індекси.

У класі valarray визначені конструктори, що дозволяють створювати порожній масив і масив, заповнений однаковими або заданими значеннями:

```
template <class T> class valarray {
public:
        // Масив нульового розміру:
Valarray ():
        // Масив з n елементів (до кожного застосовується
        // конструктор за замовчуванням:
explicit valarray (size_t n):
        // Масив з п елементів зі значенням v:
valarray (const T & v, size_t n);
        // Масив з n елементів зі значеннями з масиву т:
valarray (const T * m, size_t n);
        // Конструктор копіювання
        Valarray (const valarray &):
        // Масив зі значеннями з зрізу:
        valarray (const slice_array <T> &):
        // Масив зі значеннями з узагальненого зрізу:
        valarray (const gslice_array <T> &):
        // Масив зі значеннями з підмножини:
        valarray (const masl <_array <T> &):
        // Масив зі значеннями з indirect_array:
        valarrayCconst incl1rect_array <T> &):
...
}
```

Лекція 16. Інтеграція Python / C.

Протягом всієї книги ми працювали з програмами, написаними на мові Python. Ми користувалися інтерфейсами до зовнішніх службам і писали багато разів використовувані інструменти на мові Python, але ми використовували тільки мову Python. Незалежно від розміру і практичності наших програм вони до самого останнього символу були написані на мові Python.

Для багатьох програмістів і розробників сценаріїв це і є кінцева мета. Таке автономне програмування є одним з основних способів застосування Python. Як ми бачили, Python поставляється в повному комплекті - з інтерфейсами до системних інструментів, протоколах Інтернету, графічним інтерфейсам, сховищ даних і багатьом іншим наявним засобам. Крім того, для великого кола завдань, з якими ми зіткнемося на практиці, вже є готові рішення в світі відкритого програмного забезпечення. Система PIL, наприклад, надає можливість обробляти зображення в графічних інтерфейсах на основі бібліотеки tkinter після простого запуску майстра установки.

Але для деяких систем найважливішою характеристикою мови є можливість інтеграції (або сумісність) з мовою програмування C. Насправді роль Python як мови розширень і інтерфейсів в великих системах є однією з причин його популярності і того, що його часто називають «керуючим» мовою. Його архітектура підтримує гібридні системи, в яких змішані компоненти, написані на різних мовах програмування. Так як у кожної мови є свої сильні сторони, можливість відбирати і набирати компонент за компонентом є потужним методом. Python можна включати в усі системи, де потрібно простий і гнучкий мовний інструмент, - покладаючись на збереження швидкості виконання у випадках, коли це має значення.

Компільовані мови, такі як C і C ++, оптимізовані за швидкістю виконання, але програмувати на них складно і розробникам, і, особливо, кінцевим користувачам, якщо їм доводиться адаптувати програми. Python оптимізований по швидкості розробки, тому при використанні сценаріїв Python для управління або індивідуальної підгонки програмних компонентів, написаних на C або C ++, виходять швидкі і гнучкі системи і різко прискорюється швидкість розробки. Наприклад, перенесення окремих компонентів програм з мови Python на мову C може збільшити швидкість виконання програм. Крім того, системи, спроектовані так, що їх підгонка покладається на сценарії Python, не вимагають поставки повних вихідних текстів і вивчення кінцевими користувачами складних або спеціалізованих мов.

В цій останній технічної розділі ми коротко познайомимося з інструментами, призначеними для організації взаємодій з програмними компонентами, написаними на мові C. Крім того, ми обговоримо можливість використання Python як вбудованого мовного інструменту в інших системах і поговоримо про інтерфейси для розширення сценаріїв Python новими модулями і типами, реалізованими на мовах програмування, сумісних з C. Крім цього ми

коротко розглянемо інші технології інтеграції, менш пов'язані з мовою C, такі як Jython.

Зверніть увагу на слово «коротко» в попередньому абзаці. Через те, що далеко не всім програмістам на Python вимагається володіння цією темою; через те, що для цього потрібно вивчати мову C і порядок створення файлів з правилами складання для утиліти make; і через те, що це заключна глава і без того вельми об'ємної книги, в цьому розділі будуть опущені подробиці, які можна знайти і в комплекті стандартних посібників з мови Python, і безпосередньо у вихідних текстах самого мови Python. Замість цього ми розглянемо ряд простих прикладів, які допоможуть вам почати рухатися в цьому напрямку і продемонструють деякі можливості для систем Python.

Розширення і вбудовування. Перш ніж перейти до програмного коду, я хотів би почати з визначення, що тут мається на увазі під словом «інтеграція». Незважаючи на те, що цей термін має майже таке ж широке тлумачення, як і слово «об'єкт», нашу увагу в цьому розділі буде зосереджено на тісній інтеграції - коли передача управління між мовами здійснюється простими, прямими і швидкими викликами функцій. Раніше ми розглядали можливість організації менш тісному зв'язку між компонентами програми, використовуючи механізми взаємодії між процесами і мережеві інструменти, такі як сокети і канали, але в цій частині книги ми будемо знайомитися з більш безпосередніми і ефективними прийомами.

При об'єднанні Python з компонентами, написаними на мові C (або на інших компільованих мовах), на «верхньому рівні» може виявитися або Python, або C. З цієї причини існують дві різні моделі інтеграції і два різних прикладних інтерфейсу:

Інтерфейс розширення. Для виконання скомпільованого програмного коду бібліотек на мові C з програм Python.

Інтерфейс вбудовування. Для виконання програмного коду Python з двійкові програми C.

Розширення грає три основні ролі: оптимізація програм - перенесення частин програми на мову C часом є єдиною можливістю підвищити загальну продуктивність; використання існуючих бібліотек - бібліотеки, відкриті для використання в Python, отримують більш широке поширення; підтримка можливостей в програмах Python, які не підтримуються самим мовою безпосередньо, - програмний код на мові Python зазвичай не може звертатися до пристроїв в абсолютних адресах пам'яті, наприклад, але може викликати функції на мові C, здатні забезпечити це. Наприклад, яскравим представником розширення в дії є пакет NumPy для Python: забезпечуючи інтеграцію з оптимізованими бібліотеками реалізацій чисельних алгоритмів, він перетворює Python в гнучку і ефективну систему програмування обчислювальних задач, яку можна порівняти з Matlab.

Вбудовування зазвичай грає роль підгонки - виконуючи програмний код Python, підготовлений користувачем, система дозволяє модифікувати себе без необхідності повністю збирати заново її вихідний програмний код. Наприклад,

деякі програми реалізують проміжний рівень настройки на мові Python, який може використовуватися для підгонки програми за місцем за рахунок зміни програмного коду Python. Крім того, вбудовування іноді використовується для передачі подій обробникам на мові Python. Інструменти конструювання графічних інтерфейсів на мові Python, наприклад, зазвичай використовують прийом вбудовування для передачі призначених для користувача подій.

На рис. 16.1 наводиться схема цієї традиційної подвійний моделі інтеграції. У моделі розширення управління передається з Python в програмний код на мові C через проміжний сполучний рівень. У моделі вбудовування програмний код на мові C обробляє об'єкти Python і виконує програмний код Python, викликаючи функції Python C API. Оскільки в моделі розширення Python знаходиться «зверху», він встановлює певний порядок інтеграції, який можна автоматизувати за допомогою таких інструментів, як SWIG - генератор програмного коду, з яким ми познайомимося далі в цій главі, що відтворює програмний код сполучного рівня, необхідного для обгортання бібліотек C і C ++. Оскільки в моделі впровадження Python займає підлегле становище, при її використанні саме він надає комплект інструментів API, які можуть використовуватися програмами C.

Рис. 16.1. Традиційна модель інтеграції

У деяких випадках поділ не така чітке. Наприклад, використовуючи модуль ctypes, обговорюваний нижче, сценарії Python можуть викликати бібліотечні функції без застосування сполучного програмного коду C. У таких системах, як Cython (і її попередниці Pyrex), відмінності стають ще більш суттєвими - бібліотеки C створюються з комбінації програмного коду на мовах Python і C. У Jython і Iron-Python використовується схожа модель, тільки мова C в них заміщається мовами Java і C #, а сама інтеграція в значній мірі автоматизована. Ми познайомимося з цими альтернативами далі в цій главі. А поки зосередимося на традиційних моделях інтеграції Python / C.

У цьому розділі ми спочатку познайомимося з моделлю розширення, а потім перейдемо до дослідження основ вбудовування. Хоча ми будемо розглядати ці моделі в окремо, слід мати на увазі, що в багатьох системах обидві ці моделі комбінуються. Наприклад, вбудований програмний код Python, що виконується з C, може також використовувати приєднані розширення C для взаємодії з її утримує додатком. А в системах, заснованих на зворотних виклики,

бібліотеки C, доступні через інтерфейси розширення, в подальшому можуть використовувати техніку вбудовування для виконання обробників зворотного виклику Python.

Наприклад, при створенні кнопок за допомогою бібліотеки tkinter в Python, представленої раніше в цій книзі, ми викликали функції в бібліотеці C через прикладний інтерфейс розширення. Коли пізніше користувач натискав ці кнопки, бібліотека C реалізації інструментів графічного інтерфейсу перехоплювала події і передавала їх функцій на мові Python, використовуючи модель вбудовування. Хоча більша частина деталей прихована для програмного коду Python, тим не менш, в таких системах управління часто вільно передається між мовами. Архітектура Python відкрита і реєнтерабельним, що дозволяє з'єднувати мови довільним чином.

16.1. Розширення на C

Оскільки сам Python сьогодні написаний на C, компілюють розширення Python можуть бути написані будь-якою мовою, сумісних з C щодо стеків виклику і компонування. В їх число входять C, а також C ++ з належними оголошеннями «extern C» (автоматично поміщається в файли заголовків Python). Незалежно від мови, що використовується для реалізації розширень, існує два види розширень Python на компільованих мовах:

- **Модулі на C.**Бібліотеки інструментів, які сприймаються клієнтами як файли модулів Python.
- **Типи C.**Множинні екземпляри об'єктів, які поводяться, як вбудовані типи і класи.

Зазвичай модулі розширень на C використовуються для реалізації простих бібліотек функцій і в програмному коді Python вони з'являються у вигляді імпортованих модулів (звідси і їх назва). Типи C використовуються для програмування об'єктів, що генерують множинні екземпляри, кожен з яких має власну інформацію про стан і здатний підтримувати оператори виразів подібно до багатьох класах Python. Типи C володіють тими ж можливостями, що і вбудовані типи і класи Python. Вони володіють методами, підтримують операції додавання, індексування, вилучення зрізів і так далі.

Щоб забезпечити можливість взаємодії, як модулі, так і типи C повинні надати програмний код «сполучної шару», який буде виконувати трансляцію викликів і передавати дані між двома мовами. Цей шар реєструє свої операції в інтерпретаторі Python у вигляді покажчиків на функції C. Сполучний шар C відповідає за перетворення аргументів, переданих з Python в C, і перетворення результатів, переданих з C в Python. Сценарії Python просто експортують розширення C і користуються ними, як якщо б вони в дійсності були написані на Python. Оскільки всі дії по трансляції здійснює програмний код C, взаємодія з боку сценаріїв Python виглядає простим і прозорим.

Модулі і типи на C відповідальні також за передачу повідомлень про помилки назад в Python, при виявленні помилок, згенерованих викликами Python

API, і за управління лічильниками посилань збирача сміття для об'єктів, що зберігаються шаром C необмежено довго - об'єкти Python, збережені в програмному коді C, НЕ будуть утилізовані, поки ви забезпечите можливість зменшення лічильників посилань до нуля. Модулі та типи C можуть компонуватись з Python статично (на етапі складання) або динамічно (при першому імпортуванні). При дотриманні всіх цих умов розширення на C здатні стати ще одним інструментом, доступним для використання в сценаріях Python.

16.2. Вбудування Python в C, основні Прийоми вбудування

До сих пір в цій главі ми розглядали лише одну половину інтеграції Python / C: виклик функцій C з Python. Цей спосіб інтеграції є, мабуть, найбільш широко використовуваним - він дозволяє програмістам прискорювати виконання програм, переводячи їх на C, і використовувати зовнішні бібліотеки, створюючи для них обгортки у вигляді модулів розширень і типів на C. Але настільки ж корисним може виявитися і зворотне: виклик Python з C. Шляхом реалізації окремих компонентів програми на вбудовуваному програмному коді Python ми даємо можливість змінювати їх на місці, не поставляючи замовнику весь програмний код системи.

В даному розділі розповідається про цю другій стороні інтеграції Python / C. Тут йдеться про інтерфейси C до Python, що дозволяють програмам, написаним на C-сумісних мовах, виконувати програмний код на мові Python. В цьому режимі Python виступає в якості вбудованого керуючого мови (або, як іноді кажуть, «макромови»). Хоча вбудовування представляється тут, здебільшого, ізольовано, потрібно пам'ятати, що найкраще розглядати підтримку інтеграції в Python як єдине ціле. Структура системи зазвичай визначає відповідний підхід до інтеграції: розширення C, виклики вбудованих функцій або те й інше разом. На завершення цієї глави ми обговоримо декількох великих платформ інтеграції, таких як Jython і IronPython, які надають більш широкі можливості інтеграції компонентів.

16.2.1 Огляд API вбудовування в C

Перше, що слід відзначити в API вбудованих викликів Python, це меншу його структурованість, ніж у інтерфейсів розширення. Для вбудовування Python в C може знадобитися більш творчий підхід, ніж при розширенні: програміст повинен реалізувати інтеграцію з Python, вибираючи з усієї сукупності засобів C, а не писати програмний код, який має типову структуру. Позитивною стороною такої вільної структури є можливість об'єднувати в програмах вбудовані виклики і стратегії, створюючи довільні архітектури інтеграції.

Відсутність суворої моделі вбудовування в значній мірі є результатом менш чітко визначених цілей. При розширенні Python є чіткий поділ відповідальності між Python і C і ясна структура інтеграції. Модулі та типи C повинні відповідати моделі модулів / типів Python шляхом дотримання стандартних структур розширень. В результаті інтеграція виявляється непомітною для клієнтів Python: розширення C виглядають, як об'єкти Python, і

виконують більшу частину роботи. При цьому є додаткові інструменти, такі як SWIG, що забезпечують автоматизацію інтеграції.

Але при встановленні Python структура не так очевидна - так як зовнішнім рівнем є C, не зовсім ясно, якої моделі повинен дотримуватися вбудований код Python. У C може знадобитися виконувати завантажуються з модулів об'єкти, що завантажуються з файлів або виділені в документах рядки і так далі. Замість того щоб вирішувати, чого можна і чого не можна робити в C, Python надає набір загальних інструментів інтерфейсу вбудовування, що застосовуються і організованих згідно цілям вбудовування.

Більшість цих інструментів відповідає засобам, доступним програмам Python. В цілому, якщо можна встановити, як вирішити завдання вбудовування за допомогою чистого програмного коду Python, то, ймовірно, знайдуться кошти C API, що дозволяють досягти таких же результатів.

Так як вбудовування грунтується на виборі виклику API, знайомство з Python C API абсолютно необхідно для вирішення завдань вбудовування. У цьому розділі представлений ряд характерних прикладів вбудовування та обговорюються стандартні виклики API, але немає повного списку всіх наявних в ньому інструментів. Розібравшись з наведеними прикладами, можливо, ви звернетеся до посібникам Python по інтеграції за додатковими відомостями про те, які виклики є в цій галузі. Як уже згадувалося вище, в Python є два стандартних керівництва для програмістів, які займаються інтеграцією з C / C ++:

16.2.2.Что є вбудований код?

Перш ніж переходити до деталей, розберемося з базовими поняттями вбудовування. Коли в цій книзі йдеться про «вбудованому» програмному коді Python, мається на увазі будь-яка програмна структура мовою Python, яка може бути виконана з C допомогою безпосередньо до функції. Взагалі кажучи, вбудований програмний код Python може бути декількох видів:

1. **Рядки програмного коду**
   Програми на C можуть представляти програми Python у вигляді символьних рядків і виконувати їх як вираження або інструкції (подібно вбудованим функціям eval і exec мови Python).
2. **викликані об'єкти**
   Програми на C можуть завантажувати або звертатися до викликуваним об'єктів Python, таким як функції, методи і класи, і викликати їх зі списками аргументів (наприклад, використовуючи синтаксис Python func (* pargs, * kargs)).
3. **Файли з програмним кодом**
   Програми на C можуть виконувати цілі програмні файли Python, імпортуючи модулі і виконуючи файли сценаріїв через API або узагальнені системні виклики (наприклад, popen).

Фізично в програму C зазвичай вбудовується двоичная бібліотека Python. Програмний код Python, що виконується з C, може надходити з різноманітних джерел:

- Рядки програмного коду можуть завантажуватися з файлів, можуть бути отримані в результаті введення користувачем, з баз даних і файловхраніліщ, виділені з файлів HTML або XML, можуть читатися через сокети, будуватися або перебувати безпосередньо в програмах C, передаватися функцій розширення C з програмного коду реєстрації Python і так далі.
- Викликані об'єкти можуть завантажуватися з модулів Python, повертатися іншими викликами Python API, передаватися функцій розширення C з програмного коду реєстрації Python і так далі.
- Файли з програмним кодом просто існують у вигляді файлів, модулів і виконуваних сценаріїв.

Реєстрація є прийомом, часто використовуваних при організації зворотних викликів, який буде більш детально вивчений далі в цій главі. Але що стосується рядків програмного коду, можливих джерел існує стільки, скільки їх є для рядків символів C. Наприклад, програми на C можуть динамічно будувати довільний програмний код на мові Python, створюючи і виконуючи рядки.

Нарешті, після отримання деякого програмного коду Python, який повинен бути виконаний, необхідний якийсь спосіб зв'язку з ним: програмний код Python може зажадати передачі вхідних даних з шару C і може створити висновок для передачі результатів назад в C. Насправді вбудовування в цілому становить інтерес, коли у вбудованого програмного коду є доступ до містить його шару C. Зазвичай засіб зв'язку визначається видом вбудованого програмного коду:

- Рядки програмного коду, що є виразами Python, повертають значення виразу у вигляді вихідних даних. Крім того, як вхідні, так і вихідні дані можуть мати вигляд глобальних змінних в тому просторі імен, в якому виконується рядок програмного коду - C може встановлювати значення змінних, службовців вхідними даними, виконувати програмний код Python і отримувати змінні з результатами його виконання. Вхідні і вихідні дані можна також передавати за допомогою визо вов функцій, що експортуються розширеннями на C, - програмний код Python може за допомогою модулів або типів C отримувати або встановлювати змінні в охоплює шарі C. Схеми зв'язку часто є комбінованими. Наприклад, програмний код C може заздалегідь призначати глобальні імена об'єктів, які експортують інформацію про стан та інтерфейсні функції у вбудоенний програмний код Python.
- Викликані об'єкти можуть отримувати вхідні дані у вигляді аргументів функції і повертати результати у вигляді значень, що повертаються. Передані змінювані аргументи (наприклад, списки, словники, екземпляри класів) можна використовувати у вбудованому коді одночасно для введення і виведення - зроблені в Python зміни зберігаються в об'єктах, якими володіє C. Для зв'язку з C об'єкти можуть також використовувати техніку глобальних змінних і інтерфейсу розширень C, описану для рядків.

• Файли програмного коду здебільшого можуть використовувати для зв'язку таку ж техніку, як рядки коду. При виконанні в якості окремих програм файли можуть також використовувати прийоми взаємодій між процесами (IPC).

Природно, всі види вбудованого програмного коду можуть обмінюватися даними з C, використовуючи спільні кошти системного рівня: файли, сокети, канали і інші. Однак зазвичай ці способи діють повільніше і менш безпосередньо. В даному випадку нас як і раніше цікавить інтеграція, заснована на викликах функцій.

16.2.3. Основні прийоми вбудовування

Як можна зробити висновок з попереднього огляду, вбудовування надає велику гнучкість. Щоб проілюструвати стандартні прийоми вбудовування в дії, в цьому розділі представлений ряд коротких програм на мові C, які тим чи іншим способом виконують програмний код Python. Більшість цих прикладів використовують простий файл модуля Python, представлений в прикладі 1.

Приклад 1. PP4E \ Integrate \ Embed \ Basics \ usermod.py
(https://github.com/muxuezi/pp4p/blob/master/PP4E/Integrate/Embed/Basics/usermod.py)

```
"" "
############################################################ ##############
Програми на C виконуватимуть програмний код в цьому модулі Python в режимі вбудовування. Цей файл може змінюватися без необхідності змінювати шар C. Це звичайний, стандартний програмний код Python
(Перетворення виконуються програмою на C). Повинен знаходитися в шляху пошуку модулів Python. Програми на C можуть також виконувати програмний код модулів зі стандартної бібліотеки, таких як string.
############################################################ ######### "" "

message = 'The meaning of life ...' def transform (input):
input = input.replace ( 'life', 'Python')
return input.upper ()
```

Якщо ви хоча б мінімально знайомі з мовою Python, то помітите, що цей файл визначає рядок і функцію. Функція повертає передану їй рядок після виконання заміни в рядку і переведення всіх символів в ній в верхній регістр. З Python користуватися модулем просто:

```
... / PP4E / Integrate / Embed / Basics $ python
>>> import usermod              # Імпортувати модуль
>>> usermod.message             # Витягти рядок 'The meaning of life ...'
>>> usermod.transform (usermod.message) # викликати функцію
'THE MEANING OF PYTHON ...'
```

При належному використанні API ненабагато складніше використовувати цей модуль в C.

16.2.4. Виконує прості рядків програмного коду

Найпростіше, мабуть, запустити програмний код Python з C можна за допомогою функції API PyRun_SimpleString. Звертаючись до неї, програми C можуть виконувати програми Python, представлені у вигляді масивів символьних рядків C. Ця функція дуже обмежена: весь програмний код виконується в одному і тому ж просторі імен (модуль _ _main_ _), рядки програмного коду повинні бути інструкціями Python (не вираження), відсутня простий спосіб обміну вхідними та вихідними даними з виконуваним програмним кодом Python.

Компіляція і виконання

Щоб створити самостійно виконувану програму з цього файлу вихідного програмного коду C, необхідно скомпонувати результат його компіляції з файлом бібліотеки Python. У цьому розділі під «бібліотекою» зазвичай мається на увазі бінарний файл бібліотеки, створюваний при компіляції Python, а не стандартна бібліотека Python.

На сьогоднішній день все, що стосується Python і може знадобитися в C, компілюється в один бібліотечний файл при складанні інтерпретатора. Функція main програми надходить з програмного коду C, а в залежності від розширень, встановлених для Python, може знадобитися компоновка зовнішніх бібліотек, до яких звертається бібліотека Python.

Зрозуміло, рядки з програмним кодом на мові Python, що виконуються програмою на C, ймовірно, не варто жорстко визначати в програмі C, як показано в цьому прикладі. Замість цього їх можна завантажувати з текстового файлу або отримувати з графічного інтерфейсу, витягати з файлів HTML або XML, отримувати з бази даних або через сокети і так далі. При використанні таких зовнішніх джерел рядки з програмним кодом Python, що виконуються з C, можна довільно змінювати без необхідності перекомпілювати виконує їх програму на C. Вони можуть змінюватися на сайті або кінцевими користувачами системи. Щоб використовувати рядки програмного коду з максимальною користю, ми повинні перейти до більш гнучких інструментів API.

16.2.5. Виклик об'єктів Python

Програми на C можуть також легко працювати з об'єктами Python. Інструменти API для прямої взаємодії з об'єктами в модулі Python:

PyImport_ImportModule Імпортує модуль в C, як і раніше.
PyObject_GetAttrString Завантажує значення атрибута об'єкта по імені.
PyEval_CallObject Викликає функцію Python (або клас, або метод).
PyArg_Parse Перетворює об'єкти Python в значення C.
Py_BuildValue Перетворює значення C в об'єкти Python.

Виклик функції PyEval_CallObject в цій версії є ключовим: він виконує імпортовану функцію, передаючи їй кортеж аргументів подібно синтаксичної конструкції func (* args) в мові Python. Значення, що повертається функцією Python, надходить в C у вигляді PyObject *, узагальненого покажчика на об'єкт Python.

Приклад 1. PP4E \ Integrate \ Embed \ Basics \ embed-object.c (*https://github.com/muxuezi/pp4p/blob/master/PP4E/Integrate/Embed/Basics/embed-object.c*)

```
/ * Отримує і викликає об'єкти з модулів * /

#include <Python.h>
main () {
        char * cstr;
        PyObject * pstr, * pmod, * pfunc, * pargs;
        printf ( "embedobject \ n");
        Py_Initialize ();

        / * Отримати usermod.message * /
        mpmod = PyImport_ImportModule ( "usermod");
        pstr = PyObject_GetAttrString (pmod, "message");

        / * Перетворити рядок в C * /
        PyArg_Parse (pstr, "s", & cstr);
        printf ( "% s \ n", cstr);
        Py_DECREF (pstr);

        / * Викликати usermod.transform (usermod.message) * / pfunc = PyObject_GetAttrString (pmod, "transform"); pargs = Py_BuildValue ( "(s)", cstr);
        pstr = PyEval_CallObject (pfunc, pargs); PyArg_Parse (pstr, "s", & cstr); printf ( "% s \ n", cstr);


        / * Звільнити об'єкти * /
        Py_DECREF (pmod);
        Py_DECREF (pstr);
        Py_DECREF (pfunc); / * В main () це робити необов'язково * /
        Py_DECREF (pargs); / * Оскільки вся пам'ять і так звільняється * /
        Py_Finalize ();
        }
```

Після компіляції і виконання виходить той же результат:

```
... / PP4E / Integrate / Embed / Basics $ ./embed-object
embedobject
The meaning of life ... THE MEANING OF PYTHON ...
```

Але на цей раз весь висновок створюється засобами мови C - спочатку шляхом отримання значення атрибута message модуля Python, а потім шляхом прямого вилучення і виклику об'єкта функції transform модуля і виведення повертається їм значення, що пересилається в C. Вхідними даними функції transform є аргумент функції, а НЕ глобальна змінна, якій попередньо присвоєно значення. Зверніть увагу, що на цей раз message витягується як атрибут модуля, а не в результаті виконання рядки програмного коду з ім'ям змінної; часто є кілька способів отримати один і той же результат, використовуючи різні функції API.

Виклик функцій в модулях, як показано в цьому прикладі, дає простий спосіб організації вбудовування. Програмний код у файлі модуля можна довільно змінювати, не перекомпіліруя виконує його програму на C. Забезпечується також модель прямого зв'язку: вхідні і вихідні дані можуть мати вигляд аргументів функцій і значень.

16.2.6. Виконання рядків в словниках

Коли вище ми використовували функцію PyRun_String для виконання виразів і отримання результатів, програмний код виконувався в просторі імен наявний модуль Python. Однак іноді для виконання рядків програмного коду зручніше створювати абсолютно новий простір імен, незалежне від існуючих файлів модулів. Програма на мові C, представлена в прикладі 2, демонструє, як це робиться. Простір імен створюється як новий об'єкт словника Python, при цьому в процесі бере участь ряд нових для нас функцій API:

PyDict_New - Створює новий порожній об'єкт словника.

PyDict_SetItemString - Виконує присвоювання по ключу словника.

PyDict_GetItemString - Завантажує значення зі словника по ключу.

PyRun_String - Виконує рядок програмного коду в просторах імен, як і раніше.

PyEval_GetBuiltins - Отримує модуль з областю видимості вбудованих елементів.

Головна хитрість тут полягає в новому словнику. Вхідні і вихідні дані для рядків програмно-апаратних замінюються в цей словник при його передачі в якості словників просторів імен коду у виклику функції PyRun_String. В результаті програма на C з прикладу 2 працює в точності, як наступний програмний код Python:

```
>>> d = {}
>>> d [ 'Y'] = 2
>>> exec ( 'X = 99', d, d)
>>> exec ( 'X = X + Y', d, d)
>>> print (d [ 'X'])
101
```

Але тут кожна операція Python замінюється викликом C API.

Приклад 2. PP4E \ Integrate \ Embed \ Basics \ embed-dict.c (https://github.com/muxuezi/pp4p/blob/master/PP4E/Integrate/Embed/Basics/embed-dict.c)

```
/ * Створює новий словник простору імен для рядка програмного коду * /
#include <Python.h>

main () {
        int cval;
        PyObject * pdict, * pval;
        printf ( "embeddict \ n");
        Py_Initialize ();

        / * Створити новий простір імен * /
        pdict = PyDict_New ();
        PyDict_SetItemString (pdict, "_ _builtins_ _", PyEval_GetBuiltins ());
        PyDict_SetItemString (pdict, "Y", PyLong_FromLong (2)); / * Dict [ 'Y'] = 2
* /
        PyRun_String ( "X = 99", Py_file_input, pdict, pdict); / * Вип. інструкції. *
/
        PyRun_String ( "X = X + Y", Py_file_input, pdict, pdict); / * Той же X і Y * /
        pval = PyDict_GetItemString (pdict, "X");       / * Отримати dict [ 'X'] * /

        PyArg_Parse (pval, "i", & cval);  / * Перетворити в C * /
        printf ( "% d \ n", cval);    / * Результат = 101 * / Py_DECREF (pdict);
        Py_Finalize ();
        }
```

Після компіляції і виконання ця програма на C виведе наступне:

```
... / PP4E / Integrate / Embed / Basics $ ./embed-dict
embeddict
101
```

На цей раз висновок вийшов іншим: він відображає значення змінної Python X, присвоєне вбудованими рядками програмного коду Python і отримане в C. В цілому C може отримувати атрибути модуля, або викликаючи функцію PyObject_GetAttrString з об'єктом модуля, або звертаючись до словника атрибутів модуля за допомогою PyDict_GetItemString (також можна використовувати такі рядки виразів, але не безпосередньо). В даному випадку модуля немає взагалі, тому для звернення до простору імен програмного коду з C використовується доступ до словника за індексом.

Крім можливості розчленування просторів імен рядків з програмним кодом, незалежних від файлів модулів Python базової системи, ця схема надає природний механізм зв'язку. Значення, що зберігаються в новому словнику перед виконанням програмного коду, служать вхідними даними, а імена, якими проводиться присвоєння вбудованим програмним кодом, згодом можуть вилучатись з словника як вихідні дані. Наприклад, змінна Y у другому рядку посилається на ім'я, з яким в C присвоюється значення 2; значення змінної X присвоюється програмним кодом Python і витягується пізніше в програмі C, як результат виведення.

Тут є один тонкий прийом, що вимагає пояснення: словники, які використовуються в якості просторів імен виконуваного програмного коду, зазвичай повинні містити посилання _ _builtins_ _ на простір імен області видимості вбудованих об'єктів, яка встановлюється програмним кодом такого вигляду:

```
PyDict_SetItemString (pdict, "builtins", PyEval_GetBuiltins ());
```

Це неочевидне вимога і зазвичай для модулів і вбудованих компонентів, таких як функція exec, воно обробляється самим інтерпретатором Python. Однак в разі використання власних словників просторів імен ми повинні визначати це посилання вручну, якщо виконується програмний код повинен мати можливість посилатися на вбудовані імена. Ця вимога зберігається в Python 3.X.

Література

1. Пелешко Д.Д., Теслюк В.М. Об’єктні технології С++11: навч. Посібник. – Львів: Видавництво Львівської політехніки, 2013. – 360 с.
2. Страуструп, Б. Программирование: принципы и практика с использованием C++, 2-е изд.: Пер. с англ. - М.: ООО "И.Д. Вильямс", 2016. - 1328 с.
3. Павловская, Т.А. C/C++. Программирование на языке высокого уровня. — СПб.: Питер, 2003. - 461 с.
4. Павловская, Т.А., Щупак, Ю.А. C/C++. Структурное и объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2011. - 352 с.
5. A Repository for Libraries Meant to Work Well with the C++ Standard Library. Електронний ресурс: www.boost.org.
6. ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++ Електронний ресурс: https://isocpp.org/std/the-standard
7. ISO /IEC 9899:2011. Programming Languages - C. The C standard.
8. ISO/IEC 1 4882:2011. Programming Languages - C++. The C++ standard.
9. Кёниг, Э., Му, Б.Э. Эффективное программирование на C++. Серия C++ In-Depth, т.2.: Пер. с англ. — М.: Издательский дом "Вильямс", 2002. — 384 с.
10. Саттер, Г. Решение сложных задач на C++. Серия C++ In-Depth: Пер. с англ. — М.: Издательский дом “Вильямс”, 2008. - 400 с.
11. Страуструп, Б. Дизайн и эволюция C++: Пер. с англ. - М.: ДМК Пресс. - 448 с.
12. Уоррен, Г.С. Алгоритмические трюки для программистов. – 2-е изд.: Пер. с англ. – М.: Издательский дом "Вильямс", 2014. – 512 с.