МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

В.А.БОРОДІН

МОВА ПРОГРАМУВАННЯ С++

Навчальний посібник
для студентів механіко-математичного факультету

Київ - 2021

УДК 004.432.2
ББК
Б__

Автор:
В.А.Бородін

Рецензенти:

Рекомендовано до друку вченою радою механіко-математичного факультету
(протокол No від 2021 року)

Б76
Програмування мовою С.:
навч. посіб. / В.А.Бородін – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 200 с. ISBN (укр.)
ISBN

Висвітлено основні поняття програмування мовою С++, базові типи мови, оператори та функції й засоби створення та компіляції програм різного ступеня складності мовою С, а також елементи технології створення програмного забезпечення цією мовою. Наведено численні приклади програм і окремих фрагментів коду, що наочно ілюструють викладений матеріал.

Для студентів математичних напрямів навчання.

УДК 004.432.2
ББК 32.973.26-018.2.75
ISBN

Зміст

Вступ

Даний підручник розроблявся як продовження курсу "Мова програмування С++" для студентів другого курсу механіко-математичного факультету. Таким чином, передбачається, що читач підручника вже вивчив попередню частину курсу - "Мова програмування С" та має знання по основам програмування.

Отже, в даному підручнику не розглядаються базові типи та арифметичні операції, розгалуження та цикли, робота з масивами та всі інші теми що відносяться до тієї частини С++ яка відноситься й до мови С і таким чином передбачається перехід з програмування на мові С до мови програмування С++, тобто спочатку описуються відмінності цих мов, а потім поступово розкриваються теми по мові С++.

Даний підручник зосереджений в основному на "класичному" стандарті, тобто на стандарті С++ 98/03, хоча деякі нововведення стандарту С++11 тут теж розглянуті.

В підручнику розглядаються наступні теми: потокове введення-виведення С++, об'єктно-орієнтовне програмування на С++, робота з рядками, виключення та перетворення типів, простори імен та шаблони, стандартна бібліотека шаблонів. Таким чином автор бажав покрити всі основні відмінності С++ "старого" стандарту від мови С. Основна мета підручника - навчити писати код на С++ з використанням можливостей С++98/03 та надати основні теоретичні відомості які вимагаються від "джуніор" програміста на С++. В підручнику приводиться ілюстративний код по розглянутих темах, який можна запустити та перевірити (у випадках застосування С++11 чи інших нестандартних варіантах компіляції повідомлено в тексті).

Автор підручника сподівається, що його матеріали допоможуть читачам гарно засвоїти програмування С++ для того, щоб навчитися писати складні та великі програми та набрати базові знання які допоможуть у вивченні "нового" стандарту та/або іншої С++-подібної мови.

Домовленості стилю

Для зручності розуміння в тексті прийняті наступні домовленості.

Звичайний текст має наступний стиль:

Оскільки мова С – це компілятор, то середовище програмування повинно перетворити цю програму (або пакет програм) на файл, що може виконуватися. Для цього вона має виконати наступні дії, які звуться разом побудовою програми (building).

Тестові означення та важливі формулювання:

Програма на С (С++) – *це сукупність текстових файлів*, які зазвичай звуться **заголовними** та **вхідними** (header та source files), які містять **декларації** (declarations). **Декларації** — це визначення змінних та функцій.

Формат команд або вигляд конструкцій мови:

type variable_name = value; // англійською

чи

ім'я_типу ім'я_змінної =<значення> // українською

або

int fwrite(вказівник_на_масив, розмір_об'єкта, кількість_об'єктів, вказівник_на_файл);

Інформація яка є додатковою та необов'язковою для студента:

При додаванні цілих чисел може виходити переповнення. Проблема полягає в тому, що комп'ютер не видає попередження при їх появі: програма продовжить виконання з невірними даними. Більше того, поведінка при переповненні є визначеною стандартом та фіксованою лише для цілих без знаку (натуральних).

Приклади програмного коду на мові C/C++

```
#include <stdio.h>
int main(){
   printf("Hello, world!\n");
   getchar();  // або int c = getchar();
}
```

Результат роботи програми:

Output is: 2

Некоректні конструкції:

Якщо потрібно вказати в коді некоректну конструкцію, то це буде так:

```
//  printf("%lld\n", -9223372036854775808); // ПОМИЛКА
```

або якщо вся програма є хибною практикою:

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

int main(){
    double x, y, z;
    scanf("%f %f",&x,&y);
    z = exp(x)*cos(y);
    printf("z=%lf",z);
}
```

Повідомлення компілятору:

Error 1 error C4996: 'scanf': This function or variable may be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.

Програмний код не на мові C/C++:

```
cmake_minimum_required (VERSION 2.6)
```

```
set (PROJECT hello_world)
project (${PROJECT})
set (HEADERS hello.h)
set (SOURCES hello.cpp main.cpp)
add_executable ($ {PROJECT} $ {HEADERS} $ {SOURCES})
```

Запис командного рядку:

cmake CMakeLists.txt

або

gcc hello1.c <опції, на зразок –o hello –l<бібліотека>>

1. Вступ до С++. Відмінності С++ від С. Потокове введення-виведення С++.

Відмінності С++ від С.

Стандартні потоки вводу С++. Форматування потокового вводу та виводу. *Бібліотеки iostream, stream та їх нащадки. Бібліотека iomanip.*

Робота з файлами в потоках введення-виведення.

Булевий тип. Перевантаження функцій. Робота з посиланнями.

Видалення та видалення пам’яті за допомогою new/delete.

Основні відмінності С та С++

Мова С++ виникла як мова програмування, що доповнює мову С, тобто додає нові розширені можливості до С, при цьому дозволяючи писати програми так само як і на С. Можна сказати, що С++ є надмножиною С, тобто програми на С компілюються на С++, а ось навпаки не обов’язково. Головним чином, відмінності в С++ полягають в доданих властивостях до С++ - об'єктно-орієнтованому програмуванні, обробці винятків і також більша бібліотека функцій введення/виведення, доданих алгоритмічних бібліотеках тощо.
Головні відмінності представлені в таблицях.

Таблиця 1.1

Відмінності С та С++

С	С++
С підмножина (діалект) С++.	С++ надмножина (superset) С. Мова С запускається з С++ але С не запускає С++ код.
С має 32 ключові слова .	С++ має 52 ключові слова (до С++11).
С - процедурна мова	С++ процедурна, але й ООП мова
Перевантаження функцій та операторів не дозволено в С.	Перевантаження функцій та операторів дозволено в С++.

C	C++
C підмножина (діалект) C++.	C++ надмножина (superset) C. Мова C запускається з C++ але C не запускає C++ код.
Функції в C не визначаються в структурах.	Функції визначаються структурах та класах C++.
Немає простору імен	Простори імен використовуються в C++ для запобігання колізій.
Змінні за посиланням (Reference variables) не підтримуються C.	Змінні за посиланням (Reference variables) підтримуються C++.
Віртуальних та дружніх функцій немає в C.	Віртуальні та дружні функцій є в C++.
В C немає наслідування, інкапсуляції та поліморфізму (ООП)	C++ підтримує наслідування, інкапсуляції та поліморфізму (ООП)
C має лише malloc() та calloc() для алокації пам’яті та free() для деалокації.	C++ має оператор new та delete для алокації та деалокації пам’яті.
В C немає виключень	C++ підтримує виключення (Exception handling)
Форматоване введення/виведення scanf та printf для C.	Існують потоки введення/виведення та cin і cout для введення/виведення C++.
Функціональне програмування в чистому вигляді відсутнє в C.	З Стандарту C++11 існує можливість метапрограмування, роботи з анонімними функціями та функціонального програмування

Найголовніші відмінності

C++ повністю підтримує об'єктно-орієнтоване програмування, включаючи чотири стовпи об'єктно-орієнтованого програмування (ООП):

- Абстракція
- Інкапсуляція (Приховування даних)
- Наслідування
- Поліморфізм

Крім того, C++ містить також наступні важливі складові:

- Основна мова, що дає всі будівельні блоки, включаючи змінні, типи даних і літерали тощо. Додано методи роботи з потоками вводу-виводу, розширено можливості роботи з функціями.
- Стандартна бібліотека C++, що надає багатий набір функцій, які маніпулюють файлами, рядками тощо. До типів C додані булевий тип, клас роботи з рядками тощо.

- Стандартна бібліотека шаблонів (STL) дає багатий набір методів, які маніпулюють структурами даних тощо.
- Додано простори імен та робота з виключеннями

Робота з С файлами та створення простих програм на С++

Для того, щоб запустити програму написану на С за допомогою компілятору С++, можна нічого додатково не робити — просто відкомпілювати її як С-файл за допомогою компілятору. Однак, це вийде фактично С — код який може бути несумісний з частинами програми, які написані на С++.
Тому можна ці прості програми написати як С++ програму, створивши відповідний файл з розширенням ".cpp". В цьому файлі всі включення стандартних С-бібліотек замінюються за допомогою прибирання закінчення ".h" та додавання символу "c" до назви відповідних бібліотек.
Приклад

```
    #include <cstdio> // Link section: , бібліотека стандартного
                  // вводу-виводу С інтегрована як С++ бібліотека
    #include <cmath> // заголовний файл,
                //бібліотека математичних функцій з С в С++

    int main() // головна функція (main function): точка входу
(entry point)
    {
        float x; //визначаємо дійсну (одинарної точності) змінну
'x'
        scanf("%F",&x); // введення змінної 'x'
        double y=sin(x); /* Вираз (expression): виклик функції
sin,
                        обчислення виразу та ініціалізація
дійсної змінної (подвійною точності) 'y'     */
         printf("Result  y=%f\n",y);  //  виведення  значення
змінної y
    }
```

Це є фактично С-код, але написаний як С++ програма, яку можна скомпілювати, наприклад, за допомогою команди
g++ hello1.cpp.
Для того, щоб створити просту програму "Hello World" як С++ програму можливі ще наступні варіанти:

1. В цьому варіанті при використанні функцій та властивостей введення або виведення завжди потрібно вказувати простір імен std.

```
#include <iostream> // Бібліотека функцій введення-виведення на C++

int main(){ // точка входу (головна або драйвер функція)
  std::cout<<"Hello\n";  // команда виводу на консоль з простору  std
  std::cout<<"Hello"<<std::endl;  // команда виводу на консоль з простору  std
}
```

2. В цьому варіанті вказаний простір імен std для всієї програми та отже далі його можна не вказувати.

```
#include <iostream> // Бібліотека функцій введення-виведення на C++

using namespace std; // Вказали простір імен std для всієї програми

int main(){
   cout<<"Hello\n";// команда виводу на консоль
   cout<<"Hello"<<endl;
}
```

Примітка. До стандарту C++ 98 використовувався варіант без використання простору імен та з додаванням ".h" в файлах бібліотек. Сучасний компілятор C++ **не повинен** скомпілювати такий варіант:

```
// Варіант до  C++98
#include <iostream.h>

int main(){
   cout<<"Hello";
}
```

Введення/виведення на C++

На мові C++ дії, що пов'язані з операціями введення і виведення, виконуються за допомогою функцій бібліотек. Функції ведення і виведення бібліотек мови дозволяють читати дані з файлів та пристроїв і писати дані у файли і на пристрої. Система вводу - виводу в стандартній бібліотеці C++ реалізована у вигляді потоків. Потік вводу - виводу - це логічний пристрій, який приймає та виводить інформацію користувача. Бібліотека потоків iostream реалізована як ієрархія класів та забезпечує широкі можливості для використання оператора вводу — виводу.

На С++ введення та виведення можна робити за допомогою команд

```
std::cin>>  // команда введення
std::cout<< // команда виведення
```

Приклад:

```
#include <iostream>
int main(){
   int x;
   std::cin>>x;
   int y = x*2+1;
   std::cout<<”y=”<<y;
}
```

Стандартні потоки

Коли закінчується програма на С++,автоматично створюється чотири об'єкти, що реалізують стандартні потоки.

- cin - стандартний ввід
- cout - стандартний вивід
- cerr - стандартний вивід повідомлень про помилку
- clog - стандартний вивід повідомлень про помилку(буферизований)

Бібліотека мови С++ підтримує три рівня введення-виведення даних:

- введення-виведення потоку;
- введення-виведення нижнього рівня;
- введення-виведення для консолі і порту.

При введенні-виведенні потоку всі дані розглядаються як потік окремих байтів. Для користувача потік — це файл на диску або фізичний пристрій, наприклад, дисплей чи клавіатура, або пристрій для друку, з якого чи на який направляється потік даних. Операції введення-виведення для потоку дозволяють обробляти дані різних розмірів і форматів від одиночного символу до великих структур даних. Програміст може використовувати функції бібліотеки, розробляти власні і включати їх у бібліотеку. Для доступу до бібліотеки цих класів треба включити в програму відповідні заголовні файли.

За замовчуванням стандартні введення і виведення повідомлень про помилки відносяться до консолі користувача (клавіатури та екрана). Це означає, що завжди, коли програма очікує введення зі стандартного потоку, дані повинні надходити з клавіатури, а якщо програма виводить дані — то на екран.

У мові С++ існує декілька бібліотек, які містять засоби введення-виведення, наприклад: stdio.h, iostream.h. Найчастіше застосовують потокове введення-виведення даних, операції якого включені до складу класів istream або iostream. Доступ до бібліотеки цих класів здійснюється за допомогою використання у програмі директиви компілятора #include <iostream.h> (до С++98) або #include <iostream> (після С++98).

Для потокового введення даних вказується операція «>>» («читати з»). Це перевантажена операція, визначена для всіх простих типів і вказівника на char. Стандартним потоком введення є cin.
Формат запису операції введення має вигляд:
cin [>> values];
де *values* — змінна.
Так, для введення значень змінних x і y можна записати:
cin >> x >> y;
Кожна операція «>>» передбачає введення одного значення. При такому введенні даних необхідно дотримуватись конкретних вимог:

- для послідовного введення декількох чисел їх слід розділяти символом пропуску (« ») або Enter (дані типу char відокремлювати пропуском необов'язково);
- якщо послідовно вводиться символ і число (або навпаки), пропуск треба записувати тільки в тому випадку, коли символ (типу char) є цифрою;
- потік введення ігнорує пропуски;
- для введення великої кількості даних одним оператором їх можна розташовувати в декількох рядках (використовуючи Enter);
- операція введення з потоку припиняє свою роботу тоді, коли всі включені до нього змінні одержують значення. Наприклад, для операції введення x і y, що вказана вище, можна ввести значення x та y таким чином:

2.345 789
або
2.345
789.
Оскільки в цьому прикладі пропуск є роздільником між значеннями, що вводяться, то при введенні рядків, котрі містять пропуски у своєму складі, цей оператор не використовується. У такому випадку треба застосовувати функції getline(), get() тощо.
Для потокового виведення даних необхідна операція «<<» («записати в»), що використовується разом з ім'ям вихідного потоку cout. Наприклад, вираз
cout << x;
означає виведення значення змінної x (або запис у потік). Ця операція вибирає необхідну функцію перетворення даних у потік байтів.
Формат запису операції виведення представляється як:
cout << data [<< data1];,
де data, data1 — це змінні, константи, вирази тощо.
Потокова операція виведення може мати вигляд:
cout << "y =" << x + a – sin(x) << "\n";.
Застосовуючи логічні операції, вирази треба брати в дужки:
cout << "p =" << (a && b || c) << "\n";.

Символ переведення на наступний рядок записується як рядкова константа, тобто “\n”, інакше він розглядається не як символ керуючої послідовності, а як число 10 (код символу).

Примітка. Таких помилок можна уникнути шляхом присвоювання значення керуючих символів змінним, наприклад:

```
#define << sp " "
#define << ht "\t"
#define << hl "\n".
```

Тепер операцію виведення можна здійснити так:

```
cout << “y =” << x + a – sin(x)<< hl;
```

Слід пам’ятати, що *при виведенні даних з використанням «cout <<» не виконується автоматичний перехід на наступний рядок, для реалізації такого переходу застосовується так переведення рядка “\n” або операція endl.* Тобто, вивести рядкову константу можна, наприклад, так:

```
cout << “Виводимо речення \n“;
```

або

```
cout << “ Виводимо речення” << endl;.
```

Приклад. Програма, що містить виведення даних, пояснювальні повідомлення, а також символи переведення рядка.

```
#include <iostream> // бібліотека вводу-виводу
#include <string> // бібліотека для рядкового типу string

using namespace std; // використовуємо простір імен std для всієї програми

int main(){
 char name[] = "Петро"; // ANSI рядок name
 char middle = 'P'; // символьна змінна
 string last("Петренко"); // C++ рядок last
 int age =20; // ціла змінна
 int incentives = 2;
 float salary = 3009.75f; // дійсна змінна одиничної точності
 double percent = 8.5; // // дійсна змінна подвійної точності
 //виведення результатів
 cout << "Перевірка даних\n";
 cout << name << " "<< middle <<" " << last << "\n"<<endl;
 cout << "Вік, доплата, зарплата, відсоток:\n";
 cout << " " << age << ", " << incentives << ", " << salary << ", " << percent;
 cin.get(); // затримка – чекаємо введення символу та вводу
}
```

В останніх двох операціях виведення програми можна використати символи табуляції. Наприклад, «\t» поміщає кожне наступне ім’я або число в наступну позицію табуляції (через вісім символів), у цьому випадку маємо:

```
cout << “Вік \t доплата\t зарплата\t відсоток\t \n”;
cout << " " << age << "\t " << incentives << "\t " << salary << "\t " << percent;
```

Унаслідок того, що у першому операторі cout відсутня інструкція переведення рядка, відповідь користувача на підказку (тобто введене значення змінної prod_sum) з’явиться відразу праворуч за самою підказкою.
Використання об'єктно-орієнтованого консольного вводу-виводу з допомогою потоків (потоку) в програму необхідно включити заголовний файл <iostream>, а для файлової роботи <fstream>.

Загалом, різниця між стандартними заголовними файлами з розширенням *.h і без нього полягає в тому, що файли з розширенням *.h відносяться до мови С, а без розширення – до С++. Таким чином, програмуючи на мові С++ безпечніше використовувати заголовні файли без розширення *.h, які орієнтовані на мову С++.
Проте в цьому випадку може бути необхідним підключати додатково простори імен. При використанні стандартних бібліотек вводу- виводу таким простором імен є std.
Таким чином,

- об’єкт стандартного потоку вводу cin належить до класу istream та зв’язаний із стандартним пристроєм вводу, зазвичай клавіатурою;
- об’єкт стандартного потоку виводу cout класу ostream є зв’язаним із стандартним пристроєм виводу, зазвичай монітором;
- об’єкт cerr класу ostream, зв’язаний із стандартним пристроєм виводу повідомлень про помилки. Потоки даних, що виводяться, для об’єкту cerr є небуферизованими. Тобто кожне повідомлення, що передається в cerr, приводить до миттєвої появи повідомлень про помилки ;
- об’єкт clog класу ostream, зв’язаний із стандартним пристроєм виводу повідомлень про помилки. Потоки даних, що виводяться, для об’єкту clog є буферизованими. Тобто кожна операція повідомлення, що передається в clog, може привести до того, що вивід буде зберігатися в буфері до тиго часу поки буфер повністю не заповниться або ж поки вміст буферу не буде виведено примусово.

Вивід в потік виконується за допомогою операції «помістити в потік», а саме перевантаженої операції <<. Дана операція перевантажена для виводу елементів даних стандартних типів, для виводу рядків та значень вказівників. Операція << повертає посилання на об'єкт типу ostream, для якого вона викликана. Це

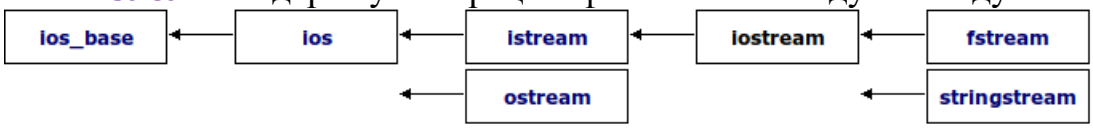
дозволяє будувати ланцюжок викликів операції «помістити в потік», що виконуються зліва направо.

Ієрархія класів та функції введення-виведення

Ієрархія класів введення-виведення достатньо складна.

Деякі класи потокового вводу - виводу.

- istream - підтримує операції по вводу;
- ostream - підтримує операції по виводу;
- iostream - підтримує операції по вводу - виводу;
- ifstream - підтримує операції вводу з файлу;
- ofstream - підтримує операції по виводу у файл;
- fstream - підтримує операції з файлами по вводу - виводу



Деякі найбільш використовувані методи:

Читання даних:

- **getline** () // читає рядок з вхідного потоку;
- **get** () // читає символ з вхідного потоку;
- **ignore** () // пропускає вказану кількість елементів від поточної позиції;
- **read** () // читає вказану кількість символів з вхідного потоку і зберігає їх в буфері (неформатований ввод).

Запис даних:

- **flush** () // очищує вміст буфера в файл (при буферизованому введенні-виведенні);
- **put** () // виводить символ в потік;
- **write** () // виводить в потік вказану кількість символів з буферу (неформатоване виведення).

Взаємність потокового і традиційного (С-стилю) введення-виведення

Деякі авторитетні керівництва по С ++ радять використовувати для вводу-виводу лише потоки STL і відмовитися від використання традиційного вводу-виводу в дусі С. Однак, це не дає змоги, наприклад, використовувати достатньо зручні специфічні властивості класичного введення-виведення. Крім того, швидкість традиційного введення виведення може бути істотно швидше. Таким чином, в реальному програмуванні С стиль вводу-виводу використовується достатньо часто. Більш того, передбачена спеціальна функція для синхронізації вводу-виводу, виконаного за допомогою поточних і старих функцій.

```
#include <iostream>
ios::sync_with_stdio(bool sync = true);
```

Можливе й навпаки заборона сумісної роботи – тобто використання лише потокового вводу:

```
ios::sync_with_stdio(false);
```

Виклик цієї процедури пришвидшує C++ введення-виведення, але унеможливлює паралельну роботу з C-вводом/виводом. Ця функція є статичним методом класу std::ios_base.
Ще одна корисна функція пришвидшення роботи з введенням-виведенням на C++:

cin.tie(NULL);

tie() - це метод, який гарантує очищення std::cout перед тим як std::cin приймає введення.
Це корисно, якщо консоль постійно приймає введення/виведення, але може сповільнювати роботу при великих обсягах даних
Для тих, хто лінується запам'ятовувати всі бібліотеки C++ (STL тощо), їх можна підключати всі одною директивою включення:

#include <bits/stdc++.h>

Тобто шаблон швидкої роботи з C++ потоками буде таким (не рекомендовано для професійного коду, скоріше для спортивного):

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    // код який використовує пришвидшене введення-виведення
    long long x=24324523, y;
    cin>>y;
    long long z = x*x;
    cout<<"y="<<y<<"\n";
}
```

Помітимо також, що cout << "\n"; працює швидше ніж cout << endl;. Команда endl працює повільніше, бо змушує обновляти потік, що не завжди потрібно.

Засоби форматування потоку

Система вводу-виводу дозволяє виконувати форматування даних та змінювати визначені параметри вводу інформації. Дані операції реалізовані за допомогою функцій форматування, прапорців та маніпуляторів.
Функції форматування є наступні:

- **width**(int wide) - дозволяє задати мінімальну ширину поля для виведення значення. При виводі задає максимальне число символів, що зчитуються. Якщо значення, що виводиться, має менше символів, ніж задана ширина поля, то воно доповнюється символами-заповнювачами до заданої ширини (за замовчуванням - пробілами). Якщо ж значення, що виводиться має більше символів, ніж ширина відведеного йому поля, то поле буде розширене.
- **precision**(int prec) - дозволяє прочитати або встановити точність (число prec) цифр після десяткової крапки), з якою виводяться числа з рухомою

крапкою. По замовчуванню числа з рухомою крапкою виводяться з точністю, рівною шести цифрам.

- **fill** (char ch) - дозволяє прочитати або встановити символ - заповнювач.

Функції cout.width(w) та cout.precision(d), які потребують підключення тільки заголовного файлу iostream.h, виконують дії, подібні тим, що і функції setw(w) та setprecision(d).
Операція введення використовує ті ж самі маніпулятори, що й операція виведення. Список змінних, в які будуть поміщені дані, визначений у values.

```
#include <iostream>
#include <cmath>
int main() {
double x;
std::cout.precision(4);
std::cout.fill('0');
std::cout << " x / Корінь(x)"<<std::endl;
for (x = 1.0; x <= 6.0; x++) {
   std::cout.width(7);
   std::cout << x << " ";
   std::cout.width(7);
   std::cout << sqrt(x) << " \n";
   }
 }
```

Результат роботи програми наступний:

```
 x / Корінь(x)
0000001 0000001
0000002 001.414
0000003 001.732
0000004 0000002
0000005 002.236
0000006 002.449
```

Опції виведення

З кожним потоком зв'язаний набір прапорців, що керують форматуванням потоку. Вони являють собою бітові маски. Встановити значення одного або кількох прапорців можна за допомогою функції-члену setf(long mask) .

- dec - Встановлюється десяткова система числення
- hex - шістнадцяткова С.Ч.
- oct - вісімкова
- scientific - числа з плаваючою крапкою(n.xxxEyy)
- showbase - виводиться основа системи числення у вигляді префіксу
- showpos - при виводі позитивних числових значень виводиться знак плюс

- uppercase - замінює нижній регістр на верхній (м - М)
- left - дані про виведення вирівнюються по лівому краю поля виводу
- right - по правому
- internal - додаються символи заповнювачі між усіма цифрами і знаками числа для заповнення поля виводу
- skipws - символи-заповнювачі (знаки пробілу, табуляції і переходу на новий рядок) відкидаються

```
#include <iostream>
using namespace std;
int main() {
 double d = 3.124e7;
 int n = 25;
 cout << "d = " << d << " ";
 cout << "n = " << n << "";
 cout.setf(std::ios_base::hex);
 cout.setf(std::ios::basefield);
 cout.setf(ios::showpos);
 cout << "d = " << d << "";
 cout << "n = " << n << "";
}
```

Результат роботи програми наступний:

```
d = 3.124e+007 n = 25 d = +3.124E+007 n = 19
```

Використання маніпулятору <iomanip>

Маніпулятори вводу-виводу являють собою вид функцій-членів класу ios, що, на відміну від звичайних функцій-членів, можуть розташовуватися усередині операцій вводу-виводу. За винятком функції-члену setw(int n), усі зміни в потоці, внесені маніпулятором, зберігаються до наступної установки. Для доступу до маніпуляторів з параметрами необхідно включити в програму стандартний заголовний файл iomanip.

- **endl** - новий рядок та очищення потоку
- **flush** - видає вміст буфера потоку у пристрій
- **setbase** (int base) - задає основу системи числення для цілих чисел(8,10,16)
- **setfill** (int c) - встановлює символ-заповнювач
- **setprecision**(int n) - встановлює точність чисел з плаваючою крапкою
- **setw** (int n) - встановлює мінімальну ширину поля виводу
- **setf**(iosbase::long mask) - встановлює ios-прапорці згідно з mask

```
int main() {
double x = 45.12345;
cout << "x = " << setprecision(4) << setfill('0') << setw(7) <<
x << endl;
}
```

Результат роботи програми наступний: x = 0045.12
Для додаткового керування даними, що виводяться, використовують маніпулятори setw(w) та setprecision(d). Маніпулятор setw(w)призначений для зазначення довжини поля, що виділяється для виведення даних (w — кількість позицій). Маніпулятор setprecision(d) визначає кількість позицій у дробовій частині дійсних чисел.
Маніпулятори змінюють вигляд деяких змінних в об'єкті cout, що у потоці розташовані за ними. Ці маніпулятори називають *прапорцями стану*. Коли об'єкт посилає дані на екран, він перевіряє прапорці, щоб довідатися, як виконати завдання, наприклад, запис:

```
cout << 456 << 789 << 123;
```

призводить до виведення значення у вигляді: 456789123, що ускладнює визначення групи значень.
Приклад Написати програму, використовуючи маніпулятор setw().

```
//  використання setw()
#include <iostream>
#include <iomanip>
using namespace std;

int main ( ){
cout << 456 << 789 << 123 << endl;
cout << setw(5) << 456 << setw(5) << 789 << setw(5) << 123 <<
endl;
cout << setw(7) << 456 << setw(7) << 789 << setw(7) << 123 <<
endl;
}
```

Результати виконання програми:

```
456789123
    456    789    123
        456       789       123
```

У цьому прикладі з'явився новий заголовний файл iomanip.h, що дозволяє застосовувати функції маніпуляторів. При використанні функції setw() число вирівнюється вправо в межах заданої ширини поля виведення. Якщо ширина недостатня, то вказане значення ігнорується.
Функція setprecision(2) повідомляє про те, що число з плаваючою крапкою виводиться з двома знаками після крапки з округленням дробової частини, наприклад, при виконанні операції
cout << setw(7) << setprecision(2) << 123.456789;
буде отримано такий результат: 123.46.

Приклад. Написати програму обчислення податку на продаж.

```
#include <iostream>
#include <iomanip>
```

```
int main ( ){
 float prod_sum; // prod_sum – сума продаж
 float nalog;
 //виведення підказки
 std::cout << "Введить суму продаж";
 std::cin >> prod_sum;
 //обчислення податку
 nalog = prod_sum * 0.7f;
 std::cout   <<   "Сума:   "   <<   std::setprecision(3)   <<
prod_sum<<std::endl;
 std::cout << "Сума без податків: " << std::setprecision(3) <<
nalog << "\n";
}
```

Результат:

```
Введить суму продаж45.56
Сума: 45.6
Сума без податків: 31.9
```

Додаткові функції введення/виведення

Для читання символу з потоку можна використовувати функцію-член get() потоку istream. Функція **get**() повертає код прочитаного символу або -1, якщо зустрівся кінець файлу вводу (ctrl/z). Функція **get**(char* str, int len, char delim) може також використовуватися для читання рядка символів. У цьому випадку використовується її варіант, у якому ця функція читає з вхідного потоку символи в буфер str, поки не зустрінеться символ-обмежувач delim (за замовчуванням –) або не буде прочитано (len-1) символів чи ознаку кінця файлу. Сам символ-обмежувач не зчитується з вхідного потоку.

Для вставки символу в потік виведення використовується функція-член **put**(char ch). Через те, що функція **get**() не читає з вхідного потоку символ-обмежувач, вона використовується не надто часто.

Набагато частіше використовується функція **getline**(char* str, int len, char delim), що читає з вхідного потоку символ-обмежувач, але не поміщає його в буфер.

Функція **gcount**() повертає число символів, прочитаних з потоку останньою операцією неформатованого вводу (тобто функцією **get**(), **getline**() або **read**()).

Розглянемо приклад, у якому використовуються дві останні функції:

```
#include <iostream>

int main() {
const size_t len = 100;
char name[len];
int count = 0;
```

```
std::cout << "Enter your name" << std::endl;
std::cin.getline(name, len);
count = std::cin.gcount();
/* Зменшуємо значення лічильника на 1, тому що getline() не
поміщає обмежувач в буфер*/
std::cout << "Number of symbols is " << count - 1 << std::endl;
}
```

Результат роботи програми наступний:

```
Enter your name
Vasya
Number of symbols is 5
```

Примітка: Оскільки для кирилиці зазвичай виколористовується двухбайтове кодування, то при спробі ввести ім'я українською отримаємо:

```
Enter your name
Вася
Number of symbols is 8
```

Для того, щоб пропустити при введенні кілька символів, використовується функція ignore(int n = l, int delim = EOF). Ця функція ігнорує n символів у вхідному потоці. Пропуск символів припиняється, якщо вона зустрічає символ-обмежувач, яким по замовчуванню є символом кінця файлу. Символ-обмежувач зчитується з вхідного потоку.

Функція peek() дозволяє "заглянути" у вхідний потік і довідатися наступний символ, що вводиться. При цьому сам символ з потоку не зчитається.

За допомогою функції putback(char ch) можна повернути символ ch у потік вводу.

Файлове введення-виведення

Робота з файлами в мові C++ як і у мові C передбачає 3 етапи: відкривання файлу (файлового потоку), обмін даними з файловим потоком, закривання файлового потоку. Для виконання операцій з файлами в мові C++ передбачено три класи: ifstream, ofstream і fstream. Ці класи є похідними від класів istream, ostream і iostream. Всі функціональні можливості (перевантажені операції << та >> для вбудованих типів, функції і прапорці форматування, маніпулятори й ін.), що застосовуються до стандартного вводу та виводу, можуть застосовуватися і до файлів. Існує деяка відмінність між використанням стандартних та файлових потоків. Стандартні потоки можуть використовуватися відразу після запуску програми, тоді як файловий потік спочатку слід зв'язати з файлом. Для реалізації файлового вводу-виводу потрібно підключити заголовний файл fstream, що знаходиться в просторі імен std.

Для введення-виводу необхідно створити потік - екземпляр відповідного класу потоку, а потім вивести його з файлу. Для потоку виводу, що використовується клас потоку, для потоку входу - ifstream, для потокового вводу-виходу - fstream. У кожному з цих класів є метод open (), який співставляє потік з файлом.

Простіше кажучи, відкриває файл. Метод передає два параметри: ім'я файлу і режим відкриття файлу. Встановлюється набір параметрів, які визначають режим відкриття файлу (запису і запису). Другий параметр необов'язковий, тобто має значення за замовченням, що відповідає класу.

- **ifstream :: open** (const char * filename, ios :: openmode mode = ios :: in);
- **ofstream :: open** (const char * filename, ios :: openmode mode = ios :: out | ios :: trunc);
- **fstream :: open** (const char * filename, ios :: openmode mode = ios :: in | ios :: out);

Всі функціональні можливості (перевизначені операції << та >> для вбудованих типів, функцій та прапорці оформлення, маніпулятори та ін.), що належать до стандартного вводу і виводу, можуть бути включені і до файлів. Існує деяка відмінність між використанням і системою потоку. Стандартні потоки можуть скористатися відразу після запуску програми, тоді як файловий потік спочатку слід зв'язати з файлом. Для релаксації файлової системи необхідно підключити заголовний файл fstream, який знаходиться в просторі std.

Таблиця 1.2

Файлові потоки

Файлові тип	
ofstream	Вихідний файл (output file stream) для створення та запису у файл
ifstream	Вхідний файл (input file stream) для читання з файлу
fstream	Файловий потік в загальному вигляді. Має властивості й ofstream та ifstream, тобто може створювати файли, записувати туди та читати інформацію звідти.

Вказані класи мають також конструктори, що дозволяють відкрити файл з буфером при створенні потоку. Параметри цих конструкторів повністю збігаються з параметрами відкриття файлу.
При помилку відкриття файлу (в контексті логічного вираження) потік отримує значення false.
Файл закривається методом **close()**. Цей метод також викликається при розбиванні екземплярів класів потоку.
Операції читання та запису в потік, пов'язаний з файлом, здійснюються з допомогою операторів << і >>, перевантажених для потоком вводу-виводу.
Відкрити файл для вводу чи виводу можна наступним чином:

```
// Для виводу
ofstream outfile;
outfile.open("File.txt");
```

або

```
ofstream outfile("File.txt");
```
або
```
fstream outfile("File.txt ",ios::out);
```
або
```
// Для вводу
ifstream infile;
infile.open("File.txt");
```
або
```
ifstream infile("File.txt");
```
або
```
fstream infile("File.txt ",ios::in);
```

Таблиця 1.3

Режими роботи з файлами

Sr.No	Прапори режимів роботи з файлами
1	ios::app - режим додавання (Append mode). Введення додається в кінець файлу
2	ios::ate — відкриває файл для виводу та ставить маркер до кінця файлу
3	ios::in — відкриває файл для читання.
4	ios::out — відкриває файл для запису.
5	ios::trunc — якщо файл існує його вміст буде збережений.

Ви можете комбінувати їх дію за допомоги логічної операції АБО (Oring). Наприклад, якщо треба відкрити файл для запису та об’єднати з випадком, коли він вже існує:
```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```
Аналогічно,можна об’єднати відкриття для читання та запису:
```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```
Режими відкриття файлу являють собою бітові маски, тому можна задавати два або більш режими, поєднуючи їх побітовою операцією АБО. Слід звернути увагу, що по замовчуванню режим відкриття файлу відповідає типові файлового потоку. У потоці вводу або виводу прапорець режиму завжди встановлений неявно. Між режимами відкриття файлу ios::ate та ios::app існує певна відмінність. Якщо файл відкривається в режимі додавання (ios::app), весь вивід у файл буде здійснюватися в позицію, що починається з поточного кінця файлу, безвідносно до операцій позиціонування у файлі. У режимі відкриття ios::ate (від англійського "at end") можна змінити позицію виводу у файл і здійснювати запис, починаючи з неї. Файли, які відкриваються для виводу, створюються, якщо вони ще не існують. Якщо при відкритті файлу не зазначений режим ios::binary, файл відкривається в текстовому режимі. Якщо відкриття файлу завершилося невдачею, об'єкт, що відповідає потокові, буде повертати нуль. Перевірити успішність відкриття файлу можна також за допомогою функції-члена is_open().

Дана функція повертає 1, якщо потік вдалося зв'язати з відкритим файлом. Для перевірки, чи досягнутий кінець файлу, можна використовувати функцію eof(). Завершивши операції вводу-виводу, необхідно закрити файл, викликавши функцію-член close(). Далі наведений приклад, що демонструє файловий ввід-вивід з використанням потоків.

Приклад. Запис та читання текстового файлу.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
 int n = 50;
  // Відкриваємо файл для виводу
  ofstream ofile("Test.txt");
  if(!ofile) {
    cout << "Файл не відкритий. ";
    return -1;
  }
  ofile << "Hello!" << n;
  // Закриваємо файл
  ofile.close();
  // Відкриваємо той же файл для вводу
  ifstream ifile("Test.txt");
  if(!ifile ) {
    cout << "Файл не відкритий.";
    return -1;
  }
  char str[80];
  ifile >> str >> n;
  cout << str << " " << n << endl;
  ifile.close(); // Закриваємо файл
}
```

Методи визначення позиції в файлі

Вказівники розташування файлів istream та ostream надають функції-члени для переміщення вказівника положення файлу. Ці функції-члени шукають ("шукати отримати") для istream і шукати ("шукати місце") для ostream.

Аргументом seekg і seekp зазвичай є довге ціле число. Другий аргумент може бути заданий для позначення напрямку пошуку. Напрямок пошуку може бути ios::beg (за замовчуванням) для позиціонування відносно початку потоку, ios::cur для позиціонування відносно поточної позиції в потоці або ios::end для позиціонування відносно кінця потік.

Вказівник положення файлу - це ціле число, яке вказує розташування у файлі як кількість байтів від початкового місця файлу. Деякі приклади позиціонування вказівника файлового положення "get" –

```
// позиція до n-го байта fileObject (передбачає ios :: beg)
fileObject.seekg (n);
// розташуємо n байтів вперед у файліObject
fileObject.seekg (n, ios :: cur);
// розміщуємо n байтів назад з кінця fileObject
fileObject.seekg (n, ios :: end);
// Позиція в кінці fileObject
fileObject.seekg (0, ios :: end);
```

Приклад. Запис та читання файлу.

```
#include <iostream>
#include <fstream>
using namespace std;
const char * filename = "testfile2.txt";
int main () {
// створення потоку, відкриття файлу для записів в текстовому режимі,
// запису даних та закриття файлу.
 ofstream ostr;
 ostr.open (filename);
 if (ostr) {
  for(int i = 0; i <16; i ++) {
    ostr << i * i << endl;
    if (ostr.bad ()) {
      cerr << "Невиправна помилка запису" << endl;
      return 1;
    }
  }
  ostr.close ();
 }
 else{
    cerr<< "Вихідний файл відкриває помилку"<< filename <<"\t";
   return 1;
 }
// відкриття файлу (в конструкторі) для читання в текстовому режимі,
// читання даних, форматоване виведення на консоль, закриття файлу.
 int data;
 int counter = 0;
 ifstream istr (filename);
 if (istr) {
```

```
   while (! (istr >> data) .eof ()) {
      if (istr.bad ()) {
          cerr << "Невиправна помилка читання" << endl;
          return 2;
      }
      cout.width (8);
      cout << data;
      if (++ counter% 4 == 0) {
         cout << endl;
      }
   }
   istr.close ();
  }
  else{
    cerr  <<  "Помилка  відкриття  вхідного  файлу"<<  filename
<<"\t";
    return 2;
  }
  return 0;
}
```

Булевий тип

В C++ булевий тип присутній відразу як базовий тип.
Значення типу приймають вигляд:
a = true;
або
b= false;

Булеві операції виконуються тими ж логічними операціями, що й в мові С.
Крім того, додається можливість виведення змінних булевого типу як в вигляді відповідного літералу, так і цілим числом за допомогою маніпуляторів *std::boolalpha, std::noboolalpha*.
Приклад.

```
#include <iostream> //cout, boolalpha, noboolalpha

int main () {
  bool a= true, b = (1==0);
  bool c = 42, d = false;
  std::cout << std::boolalpha << a << ","<< b << '\n';
  std::cout << std::noboolalpha << c << "," <<d <<'\n';
}
```

Результат:

```
true,false

1,0
```

Перевантаження функцій

Перевантаження функції є властивістю C++, що дозволяє нам створювати кілька функцій з однаковою назвою, але вони мають різні параметри.

Приклад.

```
int add(int x, int y){
return x + y;
}
```

Якщо нам потрібна таж сама функція але з дійсним аргументами, ми можемо її довизначити:

```
double add(double x, double y){
return x + y;
}
```

Або навіть визначити ту саму функцю з іншою кількістю аргументів

```
int add(int x, int y, int z){
    return x + y + z;
}
```

Примітка 1. На C такого не вдасться.

Примітка 2. А от функцію типу double add(int x, int y) вже не визначити.

***Змінна за* посиланням**

Посилання (reference) - це псевдонім, тобто інша назва для вже існуючої змінної. Після того, як посилання ініціалізовано змінною, для посилання на змінну можна використовувати ім'я змінної або посилання.

Посилання часто плутають з вказівниками, але три основні відмінності між посиланнями та вказівниками:

• Не має посилання NULL. Ви завжди повинні бути мати посилання до реальної частини пам’яті

• Після того, як посилання ініціалізовало об'єкт, його не можна змінити, щоб посилатися на інший об'єкт. Вказівники можна вказувати на інший об'єкт у будь-який час.

• Посилання має бути ініціалізовано під час його створення. Вказівники можуть бути ініціалізовані в будь-який час.

Посилання безпечніше та простіше:

1) ***Безпечніше***: Оскільки посилання повинні бути ініціалізовані, порожні посилання, такі як висячі вказівники, навряд чи існують.

2) ***Більш просте у використанні***: Оператору доступу не потрібен доступ до значення. Вони можуть використовуватися як звичайні змінні. Оператор & потрібен лише під час декларування. Крім того, до членів об'єкта можна звертатися за допомогою оператора крапки (’.’), На відміну від вказівників, де оператор стрілки (->) потрібен для доступу до членів.
Разом із зазначеними вище причинами є кілька місць, таких як аргумент конструктора копіювання, де не можна використовувати вказівник. Лише посилання може бути використано як аргумент в конструкторі копіювання. Аналогічно, посилання повинні використовуватися для перевантаження деяких операторів типу ++.

Створення посилань у C ++

Фактично ім'я змінної - це змінна, що прикріплена до розташування змінної в пам'яті. Посилання це фактично інша мітка, що прикріплена до місця розташування цієї пам'яті. Таким чином, можна отримати доступ до вмісту змінної за допомогою назви вихідної змінної або посилання. Наприклад, припустимо, що ми маємо наступну декларацію:
int x = 17;
Ми можемо оголосити змінні для посилань на x наступним чином.
int & r = x;
Тут & в цих деклараціях означає посилання. Таким чином, зчитування першого оголошення як є цілочисельним посиланням, ініціалізованим значенням «x», і зчитуванням другої декларації як "подвійне посилання ініціалізоване 17".
Приклад. Використання посилань на int та double

```
#include <iostream>

int main () {
   //декларуємо звичайні змінні
   int    i;
   double d;
   // декларуємо змінні-посилання
   int&    r = i;
   double& s = d;

   i = 5;
   std::cout << " i= " << i << std::endl;
   std::cout << " i reference = " << r  << std::endl;

   d = 11.7;
   std::cout << " d = " << d << std::endl;
   std::cout << " d reference = " << s  << std::endl;
}
```

Результат роботи

```
i= 5
 i reference = 5
 d = 11.7
 d reference = 11.7
```
Посилання використовують як аргументи-параметри та у якості того, що повертає функція.

1. Аргументи-змінні як посилання

```
// визначення функції для заміни змінних
void swap(int& x, int& y) {
   int temp;
   temp = x; /* зберігаємо значення змінної x */
   x = y;    /* покласти y в x */
   y = temp; /* покласти значення x в y */
}
```

2. Посилання як результат функції

```
#include <iostream>
#include <ctime>

using namespace std;

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};
double& setValues( int i ) {
   return vals[i];   // повертає посилання на i-ий елемент
}
// головна функція
int main () {
   cout << "Value before change" << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "vals[" << i << "] = ";
      cout << vals[i] << endl;
   }

   setValues(1) = 20.23; // змінює 2-ий елемент
   setValues(3) = 70.8;  // змінює 4-ий елемент

   cout << "Value after change" << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "vals[" << i << "] = ";
      cout << vals[i] << endl;
   }
   return 0;
}
```

Результат роботи:

```
Value before change
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1
vals[3] = 24.1
vals[4] = 50
Value after change
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 33.1
vals[3] = 70.8
vals[4] = 50
```

Примітка. Пам’ятайте що неможна повертати вказівник на тимчасово виділену змінну, тоді треба зробити її глобальною.

```
int& func() {
   int q;
   //! return q; // Помилка: Compile time error
   static int x;
   return x;     // Безпечно, x визначено й за межами функції
}
```

Оператори new та delete для виділення пам’яті на C++

Динамічне виділення пам'яті в C / C ++ це виділення пам’яті під конкретні дані вручну програмістом. Динамічно виділена пам'ять виділяється на купі, а нестатистичні та локальні змінні отримують пам'ять, виділену на стеку
Для звичайних змінних типу "int a", "char str [10]", і т.д., пам'ять автоматично виділяється і знищується. Для динамічно розподіленої пам'яті типу "int * p = new int [10]" програмісти несуть відповідальність за вилучення пам'яті, коли вона більше не потрібна. Якщо програміст не звільняє пам'ять, це викликає витік пам'яті (пам'ять не звільнюється, поки програма не завершиться).
Випадки коли потрібна динамічна пам’ять:

- динамічно розподілена пам'ять виділяє пам'ять змінної величини під масиви та структури;
- можна розподіляти та звільняти пам'ять, коли це потрібно, і коли ми більше не потребуємо. Існує багато випадків, коли ця гнучкість допомагає. Прикладами таких випадків є динамічні структури даних типу "Зв'язаний список", "Дерево" тощо.

Сі використовує функцію malloc () і calloc () для динамічного розподілу пам'яті під час виконання і використовує функцію free () для звільнення динамічно розподіленої пам'яті.

С ++ підтримує ці функції, а також два нові оператори: new і *delete*, які виконують завдання розподілу та вивільнення пам'яті краще і простіше.

Оператор new

Новий оператор позначає запит на виділення пам'яті на купі. Якщо доступна достатня кількість пам'яті, новий оператор ініціалізує пам'ять і повертає адресу знову виділеної і ініціалізованої пам'яті в змінну вказівника.

Для виділення пам’яті оператором new потрібно вказати цей оператор new, за яким слідує специфікатор типу даних і, якщо потрібна послідовність з більш ніж одного елемента, кількість їх у дужках []. Вона повертає вказівник на початок нового виділеного блоку пам'яті.

<вказівник-змінна> = new <тип даних>;

та

< вказівник-змінна> = new <тип даних> [розмір];

Щоб виділити пам'ять будь-якого типу даних одному екземплярі, використовують синтаксис:

1) **< вказівник-змінна> = new <тип даних>;**

Тут **< вказівник-змінна>** є вказівником типу **<тип даних>**. Типом даних може бути будь-який вбудований тип даних, включаючи масив або будь-які типи даних користувача, включаючи структуру та клас.

Приклад:

```
// Вказівник ініціалізується NULL
// Потім запитуються пам'ять для змінної
int * p1= NULL;
p1 = new int;
або
// Об’єднується декларація вказівника та його ініціалізація
int * p2 = new int;
```

При виконанні цих інструкцій створюються 2 об'єкти: динамічний безіменний об'єкт розміром 4 байти (значення типу int займає 4 байти) і вказівник на нього з ім'ям p1 розміром також 4 байти (у 32-ух бітній системі адреса займає 32 біти), значенням якого є адреса у пам’яті динамічного об'єкта. Можна створити й інший вказівник на той же динамічний об'єкт:

```
int *other = p1;
```

Якщо вказівникові p1 присвоїти інше значення, то можна втратити доступ до динамічного об'єкта:

```
int *ip = new int;
int i = 0;
ip = &i;
```

У результаті динамічний об'єкт як і раніше буде існувати, але звернутися до нього буде вже не можна.

Ініціалізація пам'яті.
Можна також ініціалізувати значення пам'ять за допомогою оператора new:
<вказівник-змінна> = new <тип даних> (значення);
Приклад:

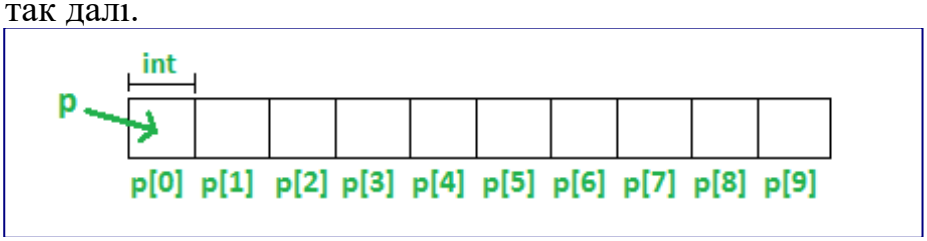
```
int * p3 = new int (25);
float * p4 = new float (75.25);
```

2) **Друга форма** (з квадратними дужками) використовується, коли потрібно виділити декілька (масив) даних певного типу.

<вказівник-змінна> = new <тип даних> [розмір]
Тут в квадратних дужках вказується кількість елементів даних типу, що виділяється в пам'яті.
Приклад:

```
int * p5 = new int [10];
або разом з ініціалізацією
int *p6 = new int[5] {1,2,3,4,};
або навіть так:
int m =10;
int * p7 = new int [m];
```

Динамічно розподіляється пам'ять на 10 цілих чисел типу int і повертає вказівник на перший елемент послідовності, який присвоюється p (вказівник).
p5[0] відноситься до першого елементу, p5[1] відноситься до другого елемента і так далі.



Примітка 1: Існує різниця між оголошенням звичайного масиву та виділенням блоку пам'яті за допомогою new. Найважливіша відмінність полягає в тому, що звичайні масиви вивільняються компілятором (якщо масив є локальним, то вивільняється, коли функція повертається або завершується). В той самий час

динамічно виділені масиви завжди залишаються там до тих пір, поки вони не будуть звільнені програмістом або програмою.
Примітка 2: Існує суттєва різниця між оголошенням нормального масиву та розподілом динамічної пам'яті для блоку пам'яті з використанням нового. Найбільш важливою відмінністю є те, що розмір регулярного масиву повинен бути постійним виразом (const expression (до C++17)) , і, отже, його розмір повинен бути визначений на момент проектування програми, перш ніж вона буде запущена, тоді як динамічне виділення пам'яті, що виконується new, дозволяє призначити пам'ять під час виконання, використовуючи будь-яке (можливо не константне) значення змінної як розмір.

Перевірка коректності виділення пам'яті
Динамічна пам'ять, запитувана нашою програмою, виділяється системою з купи пам'яті. Однак пам'ять комп'ютера є обмеженим ресурсом, і вона може бути вичерпана. Тому немає жодних гарантій, що всі запити на виділення пам'яті за допомогою оператора new будуть надані системою
Якщо в купі недостатньо пам'яті, щоб виділити, новий запит повертає відмову, викинувши виняток типу std :: bad_alloc, а якщо "nothrow" не використовується з new, і в цьому випадку він повертає порожній вказівник NULL. Таким чином, може бути гарною ідеєю перевірити змінні, створені new, перед використанням програми.

```
int *p = new(nothrow) int;
if (!p){
   cout << "Memory allocation failed\n";
}
```
або
```
//C++ 11 стиль
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // не виділена пам'ять - обробка цього випадку
}
```
При використанні підходу за допомогою nothrow, ймовірно, буде вироблений менш ефективний код, ніж при використанні виключення, оскільки він передбачає явну перевірку значення вказівника, що повертається після кожного виділення. Таким чином, механізм виключень, як правило, є кращим, принаймні для критичних ситуацій. Але оскільки ми ще не вчили обробку виключень, будемо використовувати механізм nothrow завдяки своїй простоті.

Таблиця 6.4

Відмінності new та malloc

new	malloc

Викликає конструктор	Не викликає конструктор
Оператор	Функція
Повертає змінну відповідного типу	Повертає void*
При невдачі повертає виключення std::bad_alloc	При невдачі повертає NULL
Може бути перевантаженим	Не може бути перевантаженим
Розмір рахується компілятором	Розмір рахується програмістом

***Оператор** delete*

У більшості випадків пам'ять, що виділяється динамічно, потрібна лише протягом певних періодів часу в межах програми; після того, як вона більше не потрібна, її можна звільнити, щоб пам'ять знову стала доступною для інших запитів динамічної пам'яті. Це мета оператора delete, синтаксис якої є

delete вказівник;

та

delete[] вказівник;

Перша операція вивільнює пам'ять одного елемента, виділеного за допомогою new, а друга вивільнює пам'ять, виділену для масивів елементів за допомогою new і розміру size у дужках ([]).

Приклади (звільнюємо вказівники, виділені в прикладах)

```
delete p3;
delete p4;
```

Для другого випадку:

```
  // Вивільнюємо масив на який вказує змінна p5
delete[] p5;
```

Примітка 1. Завжди звільняйте пам'ять, що виділена new (без дужок) за допомогою *delete (без дужок) та, навпаки,* new[] (з квадратними дужками) за допомогою *delete[] (з квадратними дужками). Якщо такої відповідності не буде, то є ризик або Stack Overflow або Memory Leak.*

***Примітка 2**. Видалення нульового вказівника не має ефекту, тому не потрібно перевіряти, чи є вказівник нульовий перед викликом delete.*

Операції new і delete дозволяють створювати і видаляти багатомірні динамічні масиви, підтримуючи при цьому ілюзію довільної розмірності.

Для створення динамічного двовимірного масиву використовуються наступні елементи:

- вказівник на вказівник, який містить адресу початку допоміжного масиву адрес розмір якого рівний висоті двовимірного масиву (кількості рядків);
- допоміжний масив адрес, що зберігає адреси одновимірних масивів, які власне міститимуть дані; розмір цих масивів рівний розміру ширини двовимірного масиву (кількості стовпців);
- множина масивів, що зберігають дані (реалізують рядки масиву).

Якщо вимірів більше, то використовується більша кількість допоміжних масивів, до яких приєднуватимуться інші масиви, завдяки чому власне і утворюватимуться нові виміри. Загалом можна сказати: скільки зірочок при оголошенні базового вказівника на багатовимірний масив, стільки вимірів міститиме цей масив.
Розглянемо типовий алгоритм створення динамічного багатовимірного масиву на прикладі створення динамічного двовимірного масиву елементи якого мають тип int.

Зубчаті масиви
Створюючи масиви з даними різного розміру у другому вимірі динамічних двовимірних масивів можна створювати так звані зубчаті масиви, які в окремих випадках можуть значно зекономити пам’ять в порівнянні з використанням двовимірних масивів.

```
#include <iostream>

int ** allocateTwoDimenArrayOnHeapUsingNew(int row, int col){
  int ** ptr = new int*[row];
  for(int i = 0; i < row; i++){
    ptr[i] = new int[col];
  }
  return ptr;
}

void destroyTwoDimenArrayOnHeapUsingDelete(int ** ptr, int row,
int col){
  for(int i = 0; i < row; i++){
    delete [] ptr[i];
  }
  delete [] ptr;
}

void testbyHeap(){
  int row = 2;
  int col = 3;

  int ** ptr = allocateTwoDimenArrayOnHeapUsingNew(row, col);
  ptr[0][0] = 1;   ptr[0][1] = 2;   ptr[0][2] = 3;
  ptr[1][0] = 4;   ptr[1][1] = 5;   ptr[1][2] = 6;

  for(int i = 0; i < row; i++){
```

```
    for(int j = 0; j < col; j++){
      std::cout<<ptr[i][j]<<" , ";
   }
   std::cout<<std::endl;
  }
}

int *** allocateThreeDimenArrayOnHeapUsingNew(int row, int col,
int z){
  int *** ptr = new int**[row];
  for(int i = 0; i < row; i++){
    ptr[i] = new int*[col];
    for(int j = 0; j < col; j++){
      ptr[i][j] = new int[z];
    }
  }
 ptr[0][0][0] = 1; ptr[0][0][1] = 2;
 ptr[0][1][0] = 3; ptr[0][1][1] = 4;
 ptr[0][2][0] = 5; ptr[0][2][1] = 6;
 ptr[1][0][0] = 11; ptr[1][0][1] = 12;
 ptr[1][1][0] = 13; ptr[1][1][1] = 14;
 ptr[1][2][0] = 15; ptr[1][2][1] = 16;
 return ptr;
}

int main(){

  testbyHeap();
  int *** ptr = allocateThreeDimenArrayOnHeapUsingNew(2,3,2);

  for(int i = 0; i < 2; i++){
    for(int j = 0; j < 3; j++){
      std::cout<<"(";
      for(int k = 0; k < 2; k++){
        std::cout<<ptr[i][j][k]<<",";
      }
      std::cout<<"),";
   }
  std::cout<<std::endl;
 }
}
```

2. Об’єктно-орієнтоване програмування (ООП)

Об’єктно-орієнтоване програмування. Абстракція, методи та члени класу.
Створення класів за допомогою struct та class. Інкапсуляція. Різниця між public та private.
Конструктори та деструктори.
Дружні функції, методи та класи.
Статичні методи та члени класу.

Об’єктно-орієнтований підхід полягає в наступному наборі **основних принципів**:

- Все є об’єктами.
- Всі дії та розрахунки виконуються шляхом взаємодії (обміну даних) між об’єктами, при якій один об’єкт потребує, щоб інший об’єкт виконав деяку дію. Об’єкти взаємодіють, надсилаючи і отримуючи повідомлення. Повідомлення - це запит на виконання дії, доповнений набором аргументів, які можуть знадобитися при виконанні дії.
- Кожен об’єкт має незалежну пам’ять, яка складається з інших об’єктів.
- Кожен об’єкт є представником (екземпляром, примірником) класу, який виражає загальні властивості об’єктів.
- У класі задається поведінка (функціональність) об’єкта. Таким чином усі об’єкти, які є екземплярами одного класу, можуть виконувати одні й ті ж самі дії.
- Класи організовані у єдину деревовидну структуру з загальним корінням, яка називається ієрархією успадкування. Пам’ять та поведінка, зв’язані з екземплярами деякого класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

Таким чином, програма являє собою набір об’єктів, що мають стан та поведінку. **Об’єкти** взаємодіють за допомогою використання **повідомлень**. Будується ієрархія об’єктів: програма в цілому — це об’єкт, для виконання своїх функцій вона звертається до об’єктів, що містяться у ньому, які у свою чергу виконують запит шляхом звернення до інших об’єктів програми. Звісно, щоб уникнути безкінечної рекурсії у зверненнях, на якомусь етапі об’єкт трансформує запит у повідомлення до стандартних системних об’єктів, що даються мовою та середовищем програмування. Стійкість та керованість системи забезпечуються за рахунок чіткого розподілення відповідальності об’єктів (за кожну дію відповідає певний об’єкт), однозначного означення інтерфейсів міжоб’єктної взаємодії та повної ізольованості внутрішньої структури об’єкта від зовнішнього середовища (інкапсуляції).
Основною метою створення мови програмування C++ є додавання об'єктно орієнтовних властивостей до мови програмування C. ***Таким чином, класи є***

центральною особливістю мови С++, що підтримує об'єктно-орієнтоване програмування і часто називаються користувацькими типами.
Тобто **клас** - це ***тип даних, що визначений користувачем***, має **окреслену структуру даних**, з яких складається (тобто може складатись зі стандартних типів, а може включати й інші типи даних, що визначені користувачем) та **перелік функцій,** які можуть виконувати **дії всередині класу** та *окреслено коло функцій інших класів*, що мають можливість працювати з даним типом.
Клас використовується для визначення форми об'єкта, і він поєднує представлення даних і способи маніпулювання цими даними в один акуратний пакет. Дані в класі називаються членами класу (class members). Функції в класі звуться методами класу (class methods, function methods).

Визначення класів С++

Коли ви визначаєте клас, ви визначаєте план для нового типу даних. Це насправді не визначає ніяких даних, але воно визначає ім'я класу та його структуру, тобто з чого буде складатися об'єкт класу і які операції можна виконувати на такому об'єкті.
Визначення класу починається з класу ключового слова, за яким йде ім'я класу; і тіло класу, укладене парою фігурних дужок. Визначення класу повинно супроводжуватися крапкою з комою або списком декларацій. Наприклад, ми визначили тип даних `Person` за допомогою ключового слова class наступним чином:

```
class Person {
   public: // загальний доступ до членів класу  (пояснення - далі)
      unsigned  age;   // вік
      char name[20]; // ім'я
      double height;  // зріст
      bool gender;   // стать
};// декларація класу завершується крапкою з комою
```

Насправді на С++ клас можна визначити й за допомогою ключового слова struct і новий сенс цього ключового слова - ще одна відмінність С++ від С.

```
struct Person {
   public: // загальний доступ до членів класу  (пояснення - далі)
      unsigned  age;   // вік
      char name[20]; // ім'я
      double height;  // зріст
      bool gender;   // стать
};// декларація класу завершується крапкою з комою
```

Примітка. В структурах насправді рівень доступу **public** є й по замовченню – його можна явно не писати, а ось в класах – це обов’язково (там де потрібно).

Ключове слово **public** визначає атрибути доступу членів класу, що слідують за ним. ***Загальнодоступний член*** можна отримати ***за межами класу*** в будь-якому місці в межах об'єкта класу. Ви також можете вказати члени класу як приватні або захищені.

Визначення об'єктів C++

Клас надає макет структури для об'єктів, тому в основному об'єкт створюється з класу. Конкретний екземпляр класу зветься об’єктом. Оголошуються об'єкти класу точно такою ж декларацією, що оголошуються змінні базових типів. Наступні заяви оголошують два об'єкти класу `Person`:

```
Person c1;  // Оголошуємо c1 типу Person
Person c2; // Оголошуємо c2 типу Person
```

Обидва об'єкти c1 і c2 матимуть власну копію членів даних.

Таким чином, клас - це **абстрактний тип даних**. За допомогою класу описується деяка сутність (її характеристики і можливі дії). Наприклад, клас може описувати студента, автомобіль і так далі. Описавши клас, ми можемо створити його примірник - об'єкт. **Об'єкт** - це вже **конкретний** представник класу.

Основні парадігми ООП

Для реалізації концепцій об'єктно-орієнтовного програмування (ООП) виділяють наступні поняття:

- Абстракція
- Інкапсуляція
- Наслідування
- Поліморфізм

Абстракція - дозволяє виділяти з деякої сутності тільки необхідні характеристики і методи, які повною мірою (для поставленої задачі) описують об'єкт. Наприклад, створюючи клас для опису студента, ми виділяємо тільки необхідні його характеристики, такі як ПІБ, номер залікової книжки, група. Тут немає сенсу додавати поле вагу або ім'я його кота / собаки тощо

Інкапсуляція - дозволяє приховувати внутрішню реалізацію. У класі можуть бути реалізовані внутрішні допоміжні методи (функції-члени), поля (члени класу), до яких доступ для користувача необхідно заборонити, тут і використовується інкапсуляція.

Наслідування (спадковість) - дозволяє створювати новий клас на базі іншого. Клас, на базі якого створюється новий клас, називається *базовим* (або *батьківським*), а той, що базується новий клас – *спадкоємцем* (*наслідником*). Наприклад, є базовий клас Тварина. У ньому описані загальні характеристики для всіх тварин (клас тварини, вага). На базі цього класу можна створити класи спадкоємці: Собака, Слон зі своїми специфічними властивостями. Всі властивості і методи базового класу при спадкуванні переходять в клас спадкоємець.

Поліморфізм - це здатність об'єктів з одним інтерфейсом мати різну реалізацію. Наприклад, є два класи, Круг і Квадрат. У обох класів є метод GetSquare (), який обчислює і повертає площу. Але площа кола і квадрата обчислюється по-різному, відповідно, реалізація одного і того ж методу різна.

Абстракція

Всі програми C++ складаються з наступних двох основних елементів:

• Висловлювання програми (код) - це частина програми, яка виконує дії, і вони називаються функціями.

• Дані програми - це інформація про програму, на яку впливають функції програми.

Абстракція даних є технікою програмування (і дизайну), яка спирається на поділ інтерфейсу і реалізації.

Візьмемо один реальний приклад комп'ютера, який можна вмикати і вимикати, запускати програми, регулювати гучність і додавати зовнішні компоненти, такі як динаміки, мишу та CD-програвачі, але ви не знаєте його внутрішніх деталей, ви не знаєте, як він отримує сигнали - по повітрю або через кабель, як він їх переводить, і, нарешті, відображає їх на екрані.

Таким чином, можна сказати, що інтерфейс ПК чітко відокремлює його *внутрішню реалізацію* від *зовнішнього інтерфейсу*, і ви можете грати з його інтерфейсами, як кнопка живлення, перемикач і регулятор гучності, не маючи ніяких знань про його внутрішні елементи.

У C++ класи надають великий рівень абстракції даних. Вони забезпечують достатньо загальнодоступних методів для зовнішнього світу, щоб грати з функціональністю об'єкта і маніпулювати даними об'єктів, тобто, не знаючи, як клас був реалізований внутрішньо.

Наприклад, ваша програма може зробити виклик функції sort (), не знаючи, який алгоритм фактично використовує для сортування заданих значень. Насправді, основна реалізація функцій сортування може змінюватися між випусками бібліотеки, і поки інтерфейс залишається незмінним, виклик функції буде працювати.

У C++ використовуються класи для визначення власних ***абстрактних типів даних (ADT)***. Ви можете використовувати об'єкт **cout** класу **ostream** для передачі даних у стандартний вивід, як наприклад -

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

int main() {
   cout << "Hello C++" <<endl;
   return 0;
}
```

Тут вам не потрібно розуміти, як cout відображає текст на екрані користувача. Потрібно лише знати загальнодоступний інтерфейс, а базову реалізацію "cout" можна змінювати.

У C++ використовуються ***різні рівні доступу*** для визначення абстрактного інтерфейсу для класу. Клас може містити нуль або більше міток доступу:

- Члени, позначені загальнодоступною міткою (public), які ***доступні*** для *всіх* частин програми та ***інших*** *класів*. Вигляд абстракції даних типу визначається його загальнодоступними членами.
- Члени, визначені приватною або захищеною міткою (private, protected), ***не доступні*** для коду, який використовує клас. Приватні члени приховують реалізацію від коду, який використовує цей тип.

Немає обмежень щодо частоти появи мітки доступу. Кожна мітка доступу визначає рівень доступу наступних визначень членів. Зазначений рівень доступу залишається в силі доти, доки не з'явиться наступна мітка доступу або не буде закриваюча права дужка тіла класу.

Переваги абстракції даних

Абстракція даних дає дві важливі переваги:

- Внутрішні елементи класу захищені від випадкових помилок на рівні користувача, які можуть пошкодити стан об'єкту.
- Реалізація класу може з часом змінюватися у відповідь на зміну вимог або повідомлень про помилки, не вимагаючи зміни коду рівня користувача.

Визначаючи член тільки в приватному розділі класу, автор класу може вносити зміни до даних непомітно від інших користувачів класу. Якщо реалізація змінюється, необхідно лише змінювати методи які залежать від змінених методів. В іншому ж випадку якщо дані є загальнодоступними, будь-яка функція, яка безпосередньо отримує доступ до даних старого представника, може бути порушена.

Приклад абстракції даних

Будь-яка програма C++, де реалізується клас з загальнодоступними та приватними членами, є прикладом абстракції даних. Розглянемо наступний приклад;

```
#include <iostream>
using namespace std;
class Adder {
   public:
      // Конструктор
      Adder(int i = 0) {
         total = i;
      }
      // метод - інтерфейс
      void addNum(int number) {
         total += number;
```

```
      }
      // метод - інтерфейс
      int getTotal() {
         return total;
      };

   private:
      // дані, що приховані від користувача
      int total;
};

int main() {
   Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
}
```

Коли вищезгаданий код компілюється і виконується, він дає наступний результат –

```
Total 60
```

В класі додаються числа разом, і повертається сума. Публічні члени - addNum і getTotal є інтерфейсами для зовнішнього світу, і користувачеві потрібно знати їх, щоб використовувати клас. Загальна кількість учасників - це те, про що користувач не повинен знати, але він необхідний для правильного функціонування класу.

Стратегія проектування: Абстракція розділяє код на інтерфейс і реалізацію. Таким чином, при розробці вашого компонента, ви повинні тримати інтерфейс незалежним від реалізації, так що якщо ви зміните базову реалізацію, то інтерфейс залишиться незмінним. У цьому випадку будь-які програми, які використовують ці інтерфейси не впливатимуть на роботу класів і не потребуватимуть перекомпіляції разом з ними.

Інкапсуляція

Інкапсуляція - це концепція об'єктно-орієнтованого програмування, яка пов'язує разом дані та функції, які маніпулюють даними, і яка зберігає безпеку від зовнішніх перешкод і неправильного використання. Інкапсуляція даних призвела до важливої концепції сховища даних.

Інкапсуляція даних є механізмом комплектації даних, а функції, які використовують їх і абстракцію даних, є механізмом викриття тільки інтерфейсів і приховування деталей реалізації від користувача.
C ++ підтримує властивості інкапсуляції і даних, що ховаються за допомогою створення визначених користувачем типів, які називаються класами. Отже, клас може містити приватні, захищені та загальнодоступні учасники. За замовчуванням всі елементи, визначені в класі за допомогою ключового слова class, є приватними. Наприклад:

```
class Person {
   public:
      char name[20]; // ім'я
       //  публічний вік залежить від статі
      unsigned getAge(){
     if(gender){
            return age;
     }
        else{
         return 25; // жінці завжди 25
        }
       }
   private: // захищені члени
      unsigned  age;   // вік
      bool gender;   // стать
};
```

Члени класу age, gender - мають мітку доступу **private** – приватну. Це означає, що до них можуть звертатися лише інші члени класу Box, а не будь-яка інша частина вашої програми. Це є одним із способів досягнення інкапсуляції.
Щоб зробити частину класу загальнодоступною (тобто доступною для інших частин вашої програми), ви повинні оголосити їх після ключового слова public. Всі змінні або функції, визначені після публічного специфікатора, доступні для всіх інших функцій вашої програми.

Доступ до членів даних

Доступ до відкритих членів даних об'єктів класу можна отримати за допомогою оператора прямого доступу (.). Давайте спробуємо наступний приклад, щоб зробити все зрозумілим -

```
#include <iostream>
#include <cstring>
```

```
class Person {
   public: // загальний доступ до членів класу  (пояснення - далі)
      unsigned  age;   // вік
      char name[20]; // ім'я
      double height;  // зріст
      bool gender;   // стать
};

int main() {

   Person p1,p2;
   p1.age = 21;
   p1.gender = true;
   p1. height = 78.5;
   strcpy(p1.name, "Vasya");

   p1.age = 22;
   p1.gender = true;
   p1. height = 88.5;
   strcpy(p2.name, "Petya");
std::cout<<"Average height: "<<
             (p1. height + p2.height )/2<<std::endl;
}
```

Результат:

```
Average height: 44.25
```

Важливо відзначити, що приватним і захищеним членам неможливо отримати доступ до членів класу безпосередньо за допомогою оператора прямого доступу (.) або навіть через доступ по вказівнику (оператори -> та (*).) . Для отриманя доступу до приватних і захищених членів потрібнго використати інші методи як буде показано далі.

Методи класів і об'єктів

Функція-член (метод) класу - це функція, яка має своє визначення або свій прототип у визначенні класу, як і будь-яка інша змінна. Вона діє на будь-який об'єкт класу, членом якого він є, і має доступ до всіх членів класу для цього об'єкту.

Візьмемо раніше визначений клас для доступу до членів класу, використовуючи функцію-член, замість прямого доступу до них

```
class Person {
   public:
```

```
      int  age;   // вік
      char name[20]; // ім'я
      double height;  // зріст
      bool gender;   // стать
      int getAge(); // метод отримання віку
};
```
Функції-члени можуть бути визначені в межах визначення класу або окремо за допомогою оператору визначення меж області (::) . Визначення методу в межах визначення класу оголошує функцію вбудованою (inline), навіть якщо ви не використовуєте вбудований специфікатор. Тому або ви можете визначити функцію getVolume (), як показано нижче

```
class Person {
   public:
      int  age;   // вік
      char name[20]; // ім'я
      double height;  // зріст
      bool gender;   // стать
      int getAge(){
         return gender?age:20;
      }
};
```

Можна також визначити метод getVolume за межами оператору визначення меж **(scope resolution operator) (::)** наступним чином:

```
unsigned Person::getAge(){
      return gender?age:20;
}
```
Тут важливим є те, що ви повинні використовувати ім'я класу тільки перед **оператором ::.** Функція-член буде викликана за допомогою оператора крапки (.) На об'єкті можна керувати даними, пов'язаними з цим об'єктом, лише наступним чином:

```
Person human;          // задекларувати змінну типу Person
human.getAge ();  // викликати метод getAge () з даного об’єкту
```
Приклад.

```
#include <iostream>
#include <cstring>
#include <cctype>
class Person {
   public:
      unsigned  age;   // вік
      char name[20]; // ім'я
```

```
      double height;  // зріст
      bool gender;   // стать

 // Декларація методів
      unsigned getAge();
      void setName(const char* name_);
      void setMass(double mas );
      void setAge(unsigned ag);
      void setGender(bool stat);
};

unsigned Person::getAge(){
      return gender?age:25;
}

void Person::setName(const char* name_){
    strncpy(name,name_,20);
}

void Person::setAge(unsigned ag){
    age = ag;
}

void Person::setGender(bool stat){
    gender = stat;
}

int main(){

  Person c1,c2;
  c1.setName("Vasya");
  c1.setAge(33);
  c1.setGender(true);
  std::cout<<c1.name<<" age:"<<c1.getAge()<<"\n";

  c2.setName("Masha");
  c2.setAge(29);
  c2.setGender(false);
  std::cout<<c2.name<<" age:"<<c2.getAge()<<"\n";
}
```

Результат:

```
Vasya age:33
```

```
Masha age:25
```

При цьому, як бачимо, ми не дозволяємо в певних ситуаціях узнати користувачу справжній вік Person.

Приховування даних є однією з важливих особливостей об'єктно-орієнтованого програмування, яка дозволяє запобігти безпосередньому доступу до програми внутрішнього представлення типу класу. Обмеження доступу до членів класу визначається позначеними загальнодоступними, приватними та захищеними розділами в тілі класу. Ключові слова **public**, **private** і **protected** називаються **специфікаторами доступу**.
Клас може мати кілька загальнодоступних, захищених або приватних методів та членів. Кожна секція залишається в силі доти, доки не з'явиться будь-яка інша мітка або закриється остання фігурна дужка тіла класу. Стандартний доступ для членів і класів є приватним (для класу, що визначений ключовим словом class) та публічним (для класу, що визначений ключовим словом struct) .

```
class Base {
   public:
      // загальнодоступні члени та методи
   protected:
   // приховані члени та методи
   private:
   // захищені члени та методи
};
```

Загальнодоступний член доступний з будь-якого місця поза класом, але в межах програми. Ви можете встановити та отримати значення загальнодоступних змінних без будь-якої функції-члена, як показано в наступному прикладі

```
#include <iostream>

class Number {
   public: // декларація членів та методів
      float number;
      void setNumber(float x);
      float getNumber();
};

// Визначення  методів
float Number::getNumber() {
   return number;
}

void Number::setNumber(float x) {
```

```
   number = x;
}

int main() {
   Number x;
   // встановити число за допомогою публічного  методу
   x.setNumber(5.0f);
   std::cout << "Number: " << x.getNumber() <<"\n";

   // встановити число за допомогою доступу до публічного члену
   x.number = 11.0f; // OK: because number is public
   std::cout << "Number is: " << x.number <<"\n";
}
```

Результат роботи коду:

```
Number: 5
Nimber is: 11
```

До приватної змінної чи функції члена не можна звертатися, або навіть переглядати за межами класу. До приватних учасників можуть звертатися лише функції класу та друга.

За замовчуванням всі члени класу будуть приватними, наприклад, у наступній програмі age є приватним членом класу. Отримати доступ до неї з-за меж класу можна лише за допомогою публічних методів класу

unsigned getAge(), void introduce() та void setAge(unsigned ag)

Приклад.

```
#include <iostream>
#include <cstring>

class Person {
   public:
      char name[20]; // ім'я
      bool gender;   // стать
      unsigned getAge();
      void setAge(unsigned ag);
      void introduce();
private:
      unsigned  age;   // вік
      double height;  // зріст
};

unsigned Person::getAge(){
      return gender?age:25;
}
```

```
void Person::setAge(unsigned ag){
    age = ag;
}

void Person::introduce(){
   std::cout<<"My name is "<<name<<"\n";
}

int main() {

   Person girl;
   strcpy(girl.name,"Carmen");
   girl.gender = false;
   girl.introduce();

   //girl.age = 20; // заборонена дія
   girl.setAge(25);
   std::cout<<"age is "<< girl.getAge();
}
```

Результат:

```
My name is Carmen
age is 25
```

Приклад інкапсуляції даних

Будь-яка програма C++, де реалізується клас з загальнодоступними і приватними членами, є прикладом інкапсуляції даних і абстракції даних. Розглянемо наступний приклад

```
#include <iostream>
class TimeH{
    /* private :  Ключове слово private можна не використовувати
– це специфікатор по замовченню */
    //захищені члени
    unsigned hours;
    unsigned minutes;
    //цей метод доступний лише з цього класу
    unsigned getTotalMinutes() {
        return hours*60 + minutes;
    }

public: // публічні члени та методи
   // коректна установка часу
```

```
    void setTime(unsigned h=0, unsigned m=0){
      hours = 0;
      minutes = 0;
      if(h<24 && m<60){
         hours = h;
         minutes = m;
      }
    }

    //метод для виведення часу - загальнодоступний
    void show() {
        std::cout<<"Time is:"<<hours<<":"<<minutes<<"\n";
    }
};

int main(){
  TimeH time;
  time.setTime();
  time.show();
  // time.getTotalMinutes() // заборонений доступ – буде помилка
  // time.hours = 25;

  TimeH time1;
  time1.setTime(12,20);
  time1.show();
}
```

Результат:

```
Time is:0:0
Time is:12:20
```

Даний клас зберігає дані про час. Публічні члени – setTime() і show() є інтерфейсами для зовнішнього світу, і користувач повинен знати їх, щоб використовувати клас. Закритий приватний член - це те, що приховано від зовнішнього світу, але необхідне, щоб клас працював належним чином. Таким чином, користувачу не дається можливості задати некоректний час, наприклад, просто поставивши time.hours = 25 або time.minutes = 77. Він вимушений користуватись методом setTime, що обнуляє час, коли користувач вводить некоректні дані.

Примітка. В класі зазвичай члени та методи варто робити приватними за замовчуванням, окрім тих яким повинен користуватися інший клас чи користувач. Це і є гарною інкапсуляцією. Це правило застосовується як до членів даних, так і до методів, включаючи віртуальні методи.

Конструктор класу

Конструктор класу - це **спеціальний метод класу**, що виконується, коли ми **створюємо нові об'єкти цього класу**.

Конструктор *матиме* точно таку саму ***назву***, що і ***клас***, і він взагалі ***не має типу що повертається***, навіть не має значення void. Конструктори можуть бути дуже корисними для встановлення початкових значень для певних змінних-членів.

Примітка: деякі посібники вважають, що конструктор класу неможна все ж такі називати функцією-членом класу, але на погляд автору – це суто термінологічна різниця.

Наступний приклад пояснює концепцію конструктора.

```
#include <iostream>

class Number {
   public:
      void setNumber(float x);
      double getNumber();
      Number();  // Це конструктор
   private:
      double number;
};

// Визначимо якусь дію всередині конструктора
Number::Number() {
   std::clog << "Number is being created\n";
}
void Number::setNumber( float len ) {
   number = len;
}
double Number::getNumber( void ) {
   return number;
}

int main() {
   Number num; // конструктор створює Number
   num.setNumber(3.0);
   std::cout << "Number is : " << num.getNumber() <<"\n";
}
```

Результат роботи коду:

```
Number is being created
Number is : 3
```

Примітка. Даний конкретний конструктор насправді не має особливої потреби реалізовувати, оскільки порожній непараметризований конструктор автоматично створюється при декларації класу (***конструктор за замовчуванням***).

Параметризований конструктор

Конструктор за замовчуванням не має жодного параметра, але, якщо потрібно, конструктор може мати параметри. Це допоможе вам призначити початкове значення об'єкту під час його створення, як показано в наступному прикладі

```
#include <iostream>

class Number {
   public:
      void setNumber(float x);
      float getNumber();
      Number(float x);  // Це також конструктор
   private:
      float number;
};

// Визначення конструктора та інших функцій
Number::Number(float x) {
   std::clog << "Number is being created, value = " << x <<
"\n";
   number = x;
}
/* Альтернативний варіант
Number::Number( float x): number(x) {
   cout << "Number is being created, value = " << x << "\n";
}
*/
void Number::setNumber(float x) {
   number = x;
}
float Number::getNumber() {
   return number;
}

int main() {
   Number num(15.0f); // виклик конструктору
    // отиримуємо встановлене конструктором значення
```

```
   std::cout << "Value of Number : " << num.getNumber() <<"\n";
   // змінюємо число
   num.setNumber(4.5f);
   std::cout << "Value of Number : " << num.getNumber() <<"\n";
}
```
Результат роботи програми:
```
Number is being created, value = 15
Value of Number : 15
Value of Number : 4.5
```

Альтернативний метод визначення параметризованого конструктору:
```
Number::Number( float x): number(x) {
   cout << "Number is being created, value = " << x << "\n";
}
```
Цей запис рівносильний
```
Number::Number(float x) {
   std::clog << "Number is being created, value = " << x <<
"\n";
   number = x;
}
```
Тобто для класу C, якщо у нас є наприклад члени класу X, Y, Z, etc., ми можемо використовувати наступний синтаксис: відокремлення полів та ініціалізація їх в круглих дужках:
```
C::C( double a, double b, double c): X(a), Y(b), Z(c) {
  ....
}
```
Примітка 1: Ініціалізація об’єкту при наявності параметризованого конструктору. У випадку наявності параметризованого конструктору можна використати ініціалізацію об’єкту за допомогою фігурних дужок:

```
Number m1{12}, ma{2};
C object_c{1,2,3};
```

Примітка 2: В одному класі може бути визначено декілька конструкторів (кожен з різними аргументами).

Примітка 3: Якщо в класі визначений параметризований конструктор, то конструктора за замовченням вже немає, тому якщо він все ж таки теж потрібен (наприклад для ініціалізацію порожнього масиву з об’єктів нашого класу), потрібно додати в клас і непараметризований конструктор.

Деструктор класу

Деструктор - це **спеціальна функція-член класу**, що виконується, коли об'єкт його класу виходить за межі області або коли вираз видалення (delete) застосовується до вказівника на об'єкт цього класу.

Деструктор матиме точно таку саму **назву**, що і **клас**, лише до цієї назви додається **символ тильди (~),** і він ***не може ні повертати значення, ні прийняти будь-які параметри***. Деструктор може бути дуже корисним для звільнення ресурсів перед виходом з програми, наприклад, закриття файлів, вивільнення пам'яті тощо.

Наступний приклад пояснює концепцію деструктора :

```
#include <iostream>

class Number {
   public:
      void setNumber(float x);
      float getNumber();
      Number();   // Декларація конструктору
      ~Number();  // Декларація деструктору
   private:
      float number;
};

// Визначення методів, зокрема конструктору та деструктору
Number::Number() {
   std::clog << "Object is being created\n";
}
Number::~Number() {
   std::clog << "Object is being deleted\n";
}
void Number::setNumber(float x) {
   number = x;
}
float Number::getNumber() {
   return number;
}

int main() {
   Number num;
   // встановити значення числа
   num.setNumber(42.42f);
   std::cout << "Number is : " << num.getNumber() <<"\n";
}
```

Результат роботи програми:

```
Object is being created
Number is : 42.42
Object is being deleted
```

Конструктор копіювання

Конструктор копіювання - це **конструктор**, який створює об'єкт, ***ініціалізуючи*** його ***об'єктом того ж класу***, який був створений раніше. Конструктор копіювання використовується для

- Ініціалізації одного об'єкту з іншого того ж типу.
- Копіювання об'єкту, щоб передати його як аргумент функції.
- Копіювання об'єкту, щоб повернути його з функції.

Якщо конструктор копіювання не визначений в класі, сам компілятор визначає один. Якщо клас має змінні вказівники і має деякі динамічні виділення пам'яті, то необхідно мати конструктор копіювання. Найбільш поширеною формою конструктора копіювання є така

```
classname (const classname &obj) {
  // тіло конструктору
}
```

Приклад:

```
#include <iostream>

class Number {
   public:
      float getNumber(  );
      Number( float val );             // звичайний конструктор
      Number( const Number &obj);  // конструктор копії
      ~Number();                       // деструктор
   private:
      float *ptr;
};

// Визначення методів, зокрема конструкторів та деструктору
Number::Number(float val) {
   std::cout << "Normal constructor allocating ptr\n";
   // виділяємо пам'ять під вказівник
   ptr = new float;
   *ptr = val;
}

Number::Number(const Number &obj) {
```

```
   std::clog << "Copy constructor allocating ptr.\n";
   ptr = new float;
   *ptr = *obj.ptr; // копіювання
}

Number::~Number() { // деструктор звільняє память
   std::clog << "Freeing memory!\n";
   delete ptr;
}

float Number::getNumber(  ) {
   return *ptr;
}

void display(Number obj) {
   std::cout << "Value: " << obj.getNumber() <<"\n";
}
int main() {
   Number value(10.0f);
   display(value);
}
```

Результат роботи:

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Value: 10
Freeing memory!
Freeing memory!
```

Спробуємо використати цей код в головній функції:

```
int main() {
  Number n1(10); // Конструктор
  Number n2 = n1; // Це також конструктор- копіювання
  display(n1);
  display(n2);
}
```

Результат роботи:

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Copy constructor allocating ptr.
Length of line : 10
```

```
Freeing memory!
Freeing memory!
```

Закон трьох
"Закон трьох" насправді не є законом, а скоріше орієнтиром: **якщо клас потребує явно заявленого конструктора копіювання, оператора присвоєння копії або деструктора, то зазвичай він потребує всіх трьох.**
Існують винятки з цього правила (або, скоріше, уточнення). Наприклад, іноді деструктор явно оголошується тільки для того, щоб зробити його віртуальним; у цьому випадку не обов'язково потрібно оголошувати або реалізовувати оператор копіювання і копіювання конструкторів копіювання.
Більшість класів не повинні оголошувати будь-яку з операцій "великої трійки": класи, які управляють ресурсами, взагалі потребують всіх трьох.

Дружня функція дружні методи та дружні класи

Дружня функція класу ***визначається за межами цього класу***, але ***має право доступу до всіх приватних і захищених членів*** класу. Незважаючи на те, що прототипи для дружніх функцій з'являються у визначенні класу, дружні функції не є методами класу. Така функція може бути корисною для того щоб працювати з об'єктом класу, та при цьому бути створеною за межами класу.
Дружня функція може бути ***функцією***, ***шаблоном функції*** або ***функцією-членом(методом)***, або ***шаблоном класу*** або ***класом***, у цьому випадку весь клас і всі його члени є дружними.
Щоб оголосити функцію другом класу, передуйте прототипу функції у визначенні класу ключовим словом friend наступним чином
Приклад.

```
#include <iostream>

class Person {
      unsigned  age;   // вік
public:
      void setAge (unsigned ag);
      friend void printRealAge(Person x); // вказали дружню
функцію
};

void Person::setAge(unsigned ag){
   age = ag;
}

// описали дружню функцію
void printRealAge(Person c){
   std::cout<<"Real age is "<<c.age<<"\n";
```

```
}

int main() {
    Person c;
   c.setAge(40);
   // використали дружню функцію для виведення
   printRealAge(c);
}
```
Результат:
```
Real age is 40
```

Аналогічно можна визначити і дружній метод з тією різницею, що потрібно за допомогою оператору включення(::) вказати клас та його дружній метод замість функції, як у прикладі:

```
class Person;/* нам потрібно вказати компілятору про існування цього класу перед використанням щоб не було використання класу перед його декларацією
*/
class Doctor{
public:
   void treat(Person& x);
};

class Person {
      unsigned  age;   // вік кота
public:
      void setAge (unsigned ag);
     friend void Doctor::treat(Person& x); /* вказали дружній метод класу
        - клас краще передавати за посиланням *//
};


void Doctor::treat(Person& x){
      std::cout<<"I   use   personal   data,   for   example age:"<<x.age<<"\n";
   }

int main() {
   Person c;
   //
```

```
   c.setAge(40);
   // використали дружній метод
    Doctor d;
   d.treat(c);
}
```
Результат:
```
I use personal data, for example age:40
```

Крім того, можна об’явити весь інший клас дружнім – тобто всі методи цього класу мають доступ до захищених членів та методів першого класу. За допомогою цього можна обійти в певних випадках інкапсуляцію, тому зловживати цим прийомом не варто.
Для того, щоб визначити методи класу ClassTwo дружніми методами класу ClassOne, потрібно помістити наступну декларацію в визначення класу ClassOne:
friend class ClassTwo;
Приклад:
```
/* нам потрібно вказати компілятору про існування цього класу
перед використанням щоб не було використання класу перед його
декларацією
*/
class CloseFriend;

class Person {
      unsigned  age;   // вік
public:
      void setAge (unsigned ag);
       friend class CloseFriend; // вказали дружній клас
};

class CloseFriend{
   public:
     // всі методи цього класу можуть використовувати доступ до
Person
     void talkInsideFamily(const Person& x); /* клас краще
передавати за посиланням,  краще константним, якщо ми цей клас
не змінюємо */
};

void CloseFriend::talkInsideFamily(const Person& x){
      std::cout<<"I know real age:"<< x.age<<"\n";
}
int main() {
```

```
   Person c;
   c.setAge(40);
   CloseFriend friend;
   friend.talkInsideFamily(c);
}
```

Результат роботи:

```
I know real age:40
```

Створення одного класу другом іншого викриває деталі реалізації та зменшує інкапсуляцію. Ідеальним є збереження якомога більшої кількості деталей кожного класу від усіх інших класів.

Вбудовані функції

Вбудовані (inline) функції в C++ - потужна концепція, яка зазвичай використовується з класами. Якщо функція є вбудованою, компілятор розміщує копію коду цієї функції в кожній точці, де функція викликається під час компіляції.

Будь-яка зміна вбудованої функції може вимагати перекомпіляції всіх клієнтів функції, оскільки компілятору потрібно буде замінити весь код ще раз, інакше він продовжуватиме стару функціональність.

Щоб вбудувати функцію, потрібно записати ключове слово inline перед назвою функції та визначити функцію, перед першим викликом. Компілятор може ігнорувати вбудований кваліфікатор у випадку, якщо визначена функція міститься на більш ніж одній лінії.

Визначення функції в визначенні класу є завжди вбудованим (inline), навіть без використання цього специфікатору.

Нижче наведено приклад, який використовує вбудовану функцію для повернення максимуму з двох чисел

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

inline int Max(int x, int y) {
   return (x > y)? x : y;
}

int main() {
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

Результат:

```
Max (20,10): 20
Max (0,200): 200
```

Використання вказівнику this

Сам об'єкт у C ++ має доступ до власної адреси через важливий вказівник **this**, який називається this-вказівником. Цей вказівник є неявним параметром для всіх методів класу. Таким чином, всередині методу класу він може використовуватися для посилання на об'єкт, з якого він є викликаний.

Дружні функції не можуть отримати доступ до цього посилання, тому що друзі не є членами класу. Тільки метод класу може використати цей вказівник.

```
#include <iostream>
#include <cstring>
class Person {
      unsigned  age;   // вік
      char name[20]; // ім'я
      double height;  //  зріст
      bool gender;   // стать
   public:
 // Декларація методів
      Person(unsigned a, const char* n, double h, bool g){
         strncpy(name,n,20);
         age = a;
         height = h;
         gender = g;
      }
      void getName(){
         std::cout << name<<"\t";
      }
      int compare(const Person& p) {
         return this->height > p.height;
      }
};

int main() {
   Person p1(33, "Vasya", 1.8, true);    // Визначили p1
   Person p2(27, "Petya", 1.7,true);    // Визначили p2

   if(p1.compare(p2)) {
      p2.getName();
      std::cout << " is smaller than ";
      p1.getName();
   } else {
      p2.getName();
      std::cout << "is equal to or larger than";
      p1.getName();
```

```
   }
}
```
Результат:
```
Petya is smaller than Vasya
```

Вказівник на клас

Вказівник на клас C++ використовується так само, як і вказівник на структуру, і для доступу до членів вказівника на клас, який потрібно використовувати оператор доступу ->, так само, як з вказівниками на структури. Крім того, як і для всіх вказівників, перед його використанням необхідно ініціалізувати безпосередньо сам вказівник.

Спробуємо наступний приклад, щоб зрозуміти поняття вказівника на клас

```
#include <iostream>
#include <cstring>
class Person {
      unsigned  age;   // вік
      char name[20]; // ім'я
      double height;  //
      bool gender;   // стать
   public:
      // Конструктор
      Person(unsigned a, const char* n, double h, bool g){
         strncpy(name,n,20);
         age = a;
         height = h;
         gender = g;
      }

 // Декларація методів
      void getName(){
         std::cout << name<<"\t";
      }
      unsigned getAge(){
      return gender?age:25;
      }
};

int main(void) {
Person p1(33, "Vasya", 1.8, true);    // Визначили p1
   Person p2(27, "Masha", 1.5,false);    // Визначили p2

   Person *ptrPerson ;
```

```
   ptrPerson = &p1;
// Доступ до даних через публічні методи
   std::cout << "Age of p1: " << ptrPerson->getAge() <<
std::endl;
   // Беремо вказівник – інший обєкт
   ptrPerson = &p2;
   // Отримаємо доступ до нього через вказівник
   std::cout << "Age of p2: " << ptrPerson->getAge() <<
std::endl;
}
```

Результат:

```
Age of p1: 33
Age of p2: 25
```

Статичні методи та члени

Клас може визначити статичні елементи з використанням ключового слова static. Коли ми оголошуємо член класу ***статичним***, це означає, що незалежно від того, скільки об'єктів класу створено, ***існує*** тільки ***одна копія статичного члена***.

Статичний член є спільним для ***всіх об'єктів*** класу. Всі статичні дані ініціалізуються до нуля, коли створюється перший об'єкт, якщо немає іншої ініціалізації. Ми не можемо визначити його в середині класу, але його можна ініціалізувати за межами класу, як це зроблено в наступному прикладі, повторно класифікуючи статичну змінну, використовуючи оператор дозволу меж (::) визначивши до якого класу він належить.

Спробуємо наступний приклад, щоб зрозуміти поняття статичних членів:

```
#include <iostream>
#include <cstring>
#include <string>
class Person {
      unsigned  age;   // вік
      char name[20]; // ім'я
      double height;  //  зріст
      bool gender;   // стать
   public:
      static int objectCount; // статичний член – лічильник
об’єктів
      // Конструктор
      Person(unsigned a, const char* n, double h, bool g){
         strncpy(name,n,20);
         age = a;
         height = h;
         gender = g;
```

```
         objectCount++;
      }
 // Декларація методів
      void getName(){
         std::cout << name<<"\t";
      }
      unsigned getAge(){
      return gender?age:25;
      }
};
// Інціалізуємо статичний член класу
int Person::objectCount = 0;

int main(void) {
   Person p1(33, "Vasya", 1.8, true);    // Визначили p1
   Person p2(27, "Masha", 1.5, false);    // Визначили p2
   std::cout << "Total objects: " << Person::objectCount <<
std::endl;
}
```

Результат:

```
Total objects: 2
```

Статичні методи

Оголошуючи ***метод класу статичним***, ви робите його ***незалежним від будь-якого конкретного об'єкта класу***. Статичний метод можна викликати, навіть якщо не існує жодного об'єкта класу, і до статичних функцій можна звертатися, використовуючи тільки ім'я класу і оператор визначення меж області(::). ***Статичний метод*** може отримати ***доступ тільки до статичного члена класу***, *інших статичних методів* і будь-яких інших функцій поза класом.

Статичні методи мають областю дії сам клас, і вони не мають доступ до вказівника класу this. Ви можете використовувати статичний метод, щоб визначити, чи були створені деякі об'єкти класу.

Спробуємо наступний приклад, щоб зрозуміти поняття статичних методів

```
#include <iostream>
#include <cstring>
class Person {
      unsigned  age;   // вік
      char name[20]; // ім'я
      double height;  //  зріст
      bool gender;   // стать
   public:
```

```
      static int objectCount; // статичний член – лічильник об’єктів
      // Конструктор
      Person(){}
      Person(unsigned a, const char* n, double h, bool g){
         strncpy(name,n,20);
         age = a;
         height = h;
         gender = g;
         objectCount++; // при кожній ініціалізації об’єкту лічильник збільшиться
      }

 // Декларація методів
      void getName(){
         std::cout << name<<"\t";
      }
      unsigned getAge(){
      return gender?age:25;
      }
      static bool getCount() { // статичний метод
         return objectCount<1;
      }
};

// Інціалізуємо статичний член класу
int Person::objectCount = 0;

int main(void) {
   // Друкуємо  чи загальна кількість об’єктів менше 2-х
   std::cout << "Inital Stage Count: " << std::boolalpha<<Person::getCount() << "\n";

   Person p1[3] = {Person(33,"Vasya", 1.8, true), Person(34,"Vasya2", 1.8, true), };
  // Визначили об’єкти
   // Друкуємо  чи загальна кількість об’єктів менше 2-х
   std::cout << "Final Stage Count: " << Person::getCount() << "\n";
}
```

Результат:

```
Inital Stage Count: true
```

```
Final Stage Count: false
```

3. Об’єктно-орієнтоване програмування – наслідування

Наслідування. Специфікатори доступу public, private та protected. Типи специфікаторів наслідування.
Перевантаження методів. Поліморфізм. Перевантаження унарних та бінарних операторів.
Множинне наслідування. Проблеми множинного наслідування. Ключове слово virtual.
Рядки C++ (strings). Методи класу string.

Наслідування

Однією з найважливіших концепцій об'єктно-орієнтованого програмування є наслідування (успадкування). **Наслідування *дозволяє визначити клас у термінах іншого класу***, що полегшує створення та підтримку програми. Це також дає можливість повторного використання функціональності коду та пришвидшення часу реалізації класу та модифікації класу.
При створенні класу, замість написання абсолютно нових членів даних і методів, програміст може призначити, що новий клас повинен успадковувати члени і методи (можливо не всі) існуючого класу. Цей існуючий клас називається ***базовим*** класом (***батьківським*** класом), а новий клас називається ***похідним*** класом (класом -***нащадком***).
Ідея успадкування реалізує взаємозв'язок між об’єктами, який можна назвати «належить до» або «IS-A». Наприклад, ссавець «IS-A» тварина, собака «IS-A» ссавець, отже, собака «IS-A» тварина також і так далі.

Базові та похідні класи

Клас може бути отриманий з більш ніж одного класу, що означає, що він може успадковувати дані та функції з декількох базових класів. Щоб визначити похідний клас, ми використовуємо список батьків класу для визначення базового класу. Список батьківських класів вказує один або більше базових класів і має вигляд:
КЛАС_НАСЛІДНИК: <специфікатор-доступу> БАЗОВИЙ_КЛАС
Специфікатор доступу є одним із наступних:

- загальнодоступний (public),
- захищений (protected);
- приватний (private),

а БАЗОВИЙ_КЛАС- це назва раніше визначеного класу. Якщо специфікатор доступу не використовується, то за замовчуванням він є приватним(private).
Розглянемо базовий клас **Shape** і його похідний клас **Rectangle** наступним чином

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

```
// Базовий клас
class Shape {
   public: // публічні методи
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Клас-нащадок
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;
   Rect.setWidth(5);
   Rect.setHeight(7);
   // Виведемо площу прямокутнику
   cout << "Total area: " << Rect.getArea() << endl;
   return 0;
}
```

Результат

```
Total area: 35
```

Контроль доступу та успадкування

Похідний клас може отримати доступ до всіх не приватних членів свого базового класу. Таким чином, члени базового класу, які не повинні бути доступними для функцій-членів похідних класів, повинні бути оголошені приватними в базовому класі.

Ми можемо контролювати різні типи доступу відповідно до того, хто може отримати до них доступ яким чином

*Захищені **(protected та private)** члени*

Захищені члени або методи класу дуже схожі на приватні члени або методи, але відмінність полягає в тому, що доступ до них можна отримати в класах-нащадках, які називаються похідними класами або класами-нащадками.

Це можна перевірити наступним прикладом, де є один клас нащадок CenterMass з батьківського класу Vektor. Як бачимо, клас нащадок має доступ до членів батьківського класу.

```
#include <iostream>
#include <cmath>

class Vektor {
   private:
    double z;
   protected:
      double x,y;
};

class CenterMass:Vektor { // CenterMass клас-нащадок Vektor
   double mass;
   public:
      void setCenterMass(double x, double y, double m);
      double getShoulder();
};

// Методи класу -нащадку
double CenterMass::getShoulder() {
   return mass * sqrt(x * x + y * y);
}
void CenterMass::setCenterMass(double x_, double y_, double m_)
{
   x = x_;
   y = y_;
   mass = m_;
   //z=0; // заборонить компілятор – бо це приватний член!
}
int main() {
   CenterMass x;
   // встановлюємо
   x.setCenterMass(3.0,4.0,1.0);
   std::cout << "Shoulder of Force : "<< x.getShoulder() <<
std::endl;
```

```
}
```
Результат роботи програми:
```
Shoulder of Force : 5
```

Похідний клас успадковує всі методи базового класу з такими винятками:

- конструктори, деструктори та конструктори копіювання базового класу;
- перевантажені оператори базового класу;
- дружні функції базового класу.

Тип успадкування

При виведенні класу з базового класу базовий клас може успадковуватися через **загальне**, **захищене** або **приватне** успадкування. Тип успадкування визначається специфікатором доступу, як описано вище.
На практиці достатньо рідко використовують захищене або приватне наслідування, але достатньо часто загальнодоступне наслідування.
Під час використання іншого типу спадкування застосовуються наступні правила:

- загальнодоступне наслідування **(public** inheritance) - при виведенні класу з відкритого базового класу відкриті члени базового класу стають загальнодоступними членами похідного класу, а захищені члени базового класу стають захищеними членами похідного класу. Приватні члени базового класу ніколи не доступні безпосередньо з похідного класу, але можуть бути доступні через виклики загальнодоступних і захищених методів базового класу;
- захищене наслідування **(protected** inheritance) - при виведенні з захищеного базового класу відкриті та захищені члени базового класу стають захищеними членами похідного класу;
- приватне наслідування **(private** inheritance) - при отриманні від приватного базового класу відкриті та захищені члени базового класу стають приватними членами похідного класу.

Множинне наслідування

Клас C ++ може успадкувати з більш ніж одного класу, а ось розширений синтаксис -
КЛАС_НАСЛІДНИК: специфікатор-доступу БАЗОВИЙ_КЛАС_1, специфікатор-доступу БАЗОВИЙ_КЛАС_2, ...
або
КЛАС_НАСЛІДНИК: специфікатор-доступу БАЗОВИЙ_КЛАС_1, БАЗОВИЙ_КЛАС_2, ...специфікатор-доступу БАЗОВИЙ_КЛАС_N, ...
Там, де доступ є одним, він буде наданий для кожного базового класу, ці класи будуть розділені комою, як показано вище. Спробуємо наступний приклад:

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

// Базовий клас Shape
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Базовий клас PaintCost
class PaintCost {
   public:
      int getCost(int area) {
         return area * 70;
      }
};

// Клас нащадок
class Rectangle: public Shape, public PaintCost {
   public:
      int getArea() {
         return (width * height);
      }
};

int main() {
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();
```

```
   // Вивод площі
   cout << "Total area: " << Rect.getArea() << endl;
   // Виведення вартості
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;
}
```
Результат:
```
Total area: 35
Total paint cost: $2450
```

Перезавантаження методів

C++ дозволяє вказати більше одного визначення для назви функції або оператора в деякій області. Це називається **перевантаженням функції** і **перевантаження оператору** відповідно.

Перевантажена (override) декларація - це декларація, яка оголошена з ***тією ж назвою***, що й раніше оголошена декларація в тій самій області, за винятком того, що обидві декларації мають ***різні аргументи*** і очевидно ***різні визначення*** (реалізації).

Коли ви викликаєте перевантажену функцію або оператор, компілятор визначає найбільш прийнятне визначення для використання, порівнюючи типи аргументів, які ви використовували для виклику функції або оператора з типами параметрів, вказаними в визначеннях. Процес вибору найбільш відповідної перевантаженої функції або оператора називається роздільною здатністю перевантаження.

Зокрема можна створити декілька визначень з одним іменем функції в тій же області. Визначення функції повинно відрізнятися один від одного типами та / або числом аргументів у списку аргументів. Не можна перевантажувати декларації функцій, які відрізняються тільки типом повернення.

Нижче наведено приклад, в якому для друку різних типів даних використовується одна і та ж функція print ()

```
#include <iostream>
class PrintData {
   public:
      void print(int i) {
        std::cout << "Print int: " << i << std::endl;
      }
      void print(float f) {
        std::cout << "Print float: " << f << std::endl;
      }
      void print(double df) {
        std::cout << "Printdouble: " << df << std::endl;
      }
```

```
      void print(const char* c) {
        std::cout << "Print characters: " << c << std::endl;
      }
};

int main(void) {
   PrintData pd;
   // Виклик методу print для цілого
   pd.print(5);
   // Виклик методу print для float
   pd.print(25.35f);
   // Виклик методу print для double
   pd.print(234.567);
// Виклик методу print для символів
   pd.print("Hello C++");
}
```

Результат:

```
Print int: 5
Print float: 25.35
Print double: 234.567
Print characters: Hello C+
```

Перевантаження операторів в C ++

Ви можете перевизначити або перевантажити більшість вбудованих операторів, доступних у C++. Таким чином, програміст може також використовувати оператори з визначеними користувачем типами.

Перевантажені оператори - це функції зі спеціальними іменами: ключове слово "operator", за яким слідує символ, який визначається оператором. Як і будь-яка інша функція, перевантажений оператор має тип повернення і список параметрів.

Приклад.

Person operator+ (const Person &);

Ця декларація оголошує оператор додавання, який можна використовувати для додавання двох об'єктів Person і повертає кінцевий об'єкт Person. Більшість перевантажених операторів можуть бути визначені як звичайні функції (які не є методами якогось класу) або функції-члени класу. У випадку, якщо ми визначимо вищезгадану функцію як нечленовану функцію класу, то нам доведеться передати два аргументи для кожного операнду наступним чином

Person operator+ (const Person &, const Person &);

Нижче наведено приклад, що показує концепцію перевантаження оператору за допомогою функції-члена. Тут об'єкт передається як аргумент, властивості якого будуть доступні за допомогою цього об'єкта, об'єкт, який викликатиме цей

оператор, може бути доступний за допомогою цього оператора, як описано нижче

```
#include <iostream>
#include <cmath>

class Vektor {
   public:
      double getLength() {
         return sqrt(x*x + y*y);
      }
      void setX(double x_) {
         x = x_;
      }
      void setY(double y_) {
         y = y_;
      }
      // Overload + operator to add two Vektor objects.
      Vektor operator+(const Vektor& b) {
         Vektor res;
         res.x = this->x + b.x;
         res.y = this->y + b.y;
         return res;
      }
   private:
      double x;
      double y;
};
int main() {
   Vektor v1, v2;              // Дві змінні класу вектор
   Vektor v3;               // Змінна класу вектор для результату
   double len = 0.0;      // Змінна для довжини
   // ініціалізація v1
   v1.setX(6.0);
   v1.setY(3.0);
   // ініціалізація v2
   v2.setX(6.0);
   v2.setY(2.0);
   // розмір v1
   len = v1.getLength();
   std::cout << "Length of v1: " << len <<"\n";
   // розмір v2
   len = v2.getLength();
```

```
    std::cout << "Length of v2: " << len <<"\n";
    // Додати два обєкти перевантеженим додаванням
    v3 = v1 + v2;
    // розмір v3
    len = v3.getLength();
    std::cout << "Length of v3 : " << len<<"\n";
}
```

Результат роботи програми:

```
Length of v1: 6.7082
Length of v2: 6.32456
Length of v3 : 13
```

Оператори які можна та неможна перевантажувати

Список операторів, які можна перевантажувати:

+ , -, *, /, % ,^ , & , |, ~, !, , , =,<,>, <=, >=, ++, –, <<, >>, ==, !=, &&, ||, +=, -=, /= ,%=, ^=, &=, |= ,*=, <<=, >>=, [], (), ->, ->*, new, new [], delete, delete []

А ось оператори які не можна перевантажувати:

::, .*, . , ?:

Приклади перевантаження оператора

Ось різні приклади перевантаження оператора, які допоможуть вам зрозуміти концепцію.

Одинарні оператори працюють на одному операнді і наступні приклади операторів:

- Оператори інкременту (++) і декременту (--).
- Унарний мінус (-) оператор.
- Логічний оператор заперечення (!).

Одинарні оператори працюють з об'єктом, так само як і з звичайною змінною, тобто ці оператори з'являється на лівій стороні об'єкта як !obj або на правій як obj--.

Наступний приклад пояснює, як може бути перевантажений оператор мінус (-):

```
#include <iostream>
class Vektor {
      double x,y;
     public:
     Vektor(double x_, double y_):x(x_),y(y_) {}
      // перевантажений (overloaded) minus (-) оператор
        Vektor operator- () {
          x = -x;
          y = -y;
```

```
          return Vektor(x,y);
        }
      void showVektor(){
          std::cout<<"Vektor is ("<<x<<","<<y<<")\n";
      }
};
```
Результат:
```
Vektor is (-2,3)
```

Ще один приклад, в якому той самий оператор перевантажується трохи інакше та перевантажуються префіксний та постфіксний декремент:

```
#include <iostream>

class Angle {
   private:
      int gradus;          // Градус 0-359
      int minutes;         // хвилина 0-60
   public:
      // конструктор 1
      Angle() {
         gradus = 0;
         minutes = 0;
      }
     // конструктор 2
      Angle(int g, int m) {
         gradus = g % 360;
         minutes = m % 60;
      }
      // відображення відстані
      void displayAngle() {
         std::cout << "Angle is " << gradus << " g and " <<
minutes <<"'\n";
      }
      // перевантажений (overloaded) minus (-) оператор
      Angle operator- () {
         gradus = -gradus;
         return Angle(gradus, minutes);
      }
   // префіксний та постфіксний декремент
   Angle& operator--(){       // префіксний декремент
       gradus--;
```

```
        return *this;
    }
    Angle operator--(int){     // // постфіксний декремент
        Angle temp = *this;
        --*this;
        return temp;
    }

};
int main() {
     Angle a1(45, 30), a2(-5, 15);
    -a1;                       // застосуємо оператор -
    a1.displayAngle();    // покажемо a1

    -a2;                       // застосуємо оператор -
    a2.displayAngle();    // покажемо a2
       a2--;
    a2.displayAngle();
    (--a3).displayAngle()
}
```

Результат роботи:

```
Angle is -45 g and 30'
Angle is 5 g and 15'
Angle is 4 g and 15'
Angle is -41 g and 45'
```

Бінарні оператори (Binary operators) приймають два аргументи. Частіше за все це оператори додавання (+) , віднімання (-) , множення(*) та ділення (/). Можемо, наприклад додати перевантаження оператору (+) в попередній клас.

```
      Angle operator+(const Angle& a) {
        int p=0;
        int m = this->minutes + a.minutes;
        if(m>=60) {
         p=1;
            minutes -= 60;
     }
        int g = gradus + a.gradus + p;
        return Angle(g, m);
      }
```

```
int main() {
    Angle a1(45, 30), a2(-5, 15);
   -a1;                        // застосуємо оператор -
   a1.displayAngle();    // покажемо a1

   -a2;                        // застосуємо оператор -
   a2.displayAngle();    // покажемо a2

   Angle a3 = a1 + a2;
   a3.displayAngle();
}
```
Результат:
```
Angle is -45 g and 30'
Angle is 5 g and 15'
Angle is -40 g and 45'
```

Аналогічно можна перевантажувати оператори (-) , ділення (/), оператори порівняння та інші бінарні оператори.
Приклад:

```
#include <iostream>
#include <cmath>

class Vektor {
   double x;      // координати вектору
   double y;     //

   public:

   Vektor(){}
   Vektor(double x_, double y_):x(x_),y(y_) {}

   double length() {
      return sqrt(x*x + y*y);
   }

   void setX(double x){
      this->x = x;
   }
   void setY(double x){
      this->y = x;
```

```
   }
   // Перевантаження оператору мінус
   Vektor operator-(const Vektor& b) {
      Vektor res;
      res.setX(this->x - b.x);
      res.setY(this->y - b.y);
      return res;
   }

  // Перевантаження оператору порівняння
   bool operator<(Vektor& b) {
      return length()<b.length();
   }

};

int main() {
   Vektor v1(1.0,1.0),v2,v3;
   double len = 0.0;

   // v2 ініціалізація
   v2.setX(2.0);
   v2.setY(3.0);
   std::cout << "v1<v2 is "<<std::boolalpha<<(v1<v2)<<"\n";
   // Різниця двох об’єктів класу:
   v3 = v1 - v2;
    // Довжина v3
   std::cout << "| v3 | = " << v3.length() <<"\n";
}
```

Результат роботи:

```
v1<v2 is true
| v3 | = 2.2360
```

Перевизначення операторів введення-виведення

Ще одним важливим варіантом перевизначення операторів є можливість перевизначення операторів >> та <<. Це дозволить використати їх задля введення та виведення в потоках cin, cout. При цьому слід пам’ятати, що потрібно надати стандартним класам можливість доступу до даних (зокрема, захищених) нашого класу. Це можна зробити за допомогою ключового слова friend.

```
#include <iostream>
#include <cmath>
```

```
class Vektor {
   double x;       // координати вектору
   double y;     //
      public:

   Vektor(){}
   Vektor(double x_, double y_):x(x_),y(y_) {}
// ****

   friend  std::ostream&  operator<<(std::ostream&  os,  const
Vektor& v){
     os<<"("<<v.x<<","<<v.y<<")";
     return os;
   }
   friend  std::istream&  operator>>(std::istream&  is,  Vektor&
v){
     is>>v.x>>v.y;
     return is;
   }
};

int main() {
   Vektor v1,v2,v3;
   std::cout<<"v1:";
   std::cin>>v1;
   std::cout<<"v2:";
   std::cin>>v2;
   // Додати два об’єкти класу:
   v3 = v1 - v2;
   // Вивести результат:
   std::cout<<v3<<"\n";
}
```

Результат: (вводимо перший вектор 2.3 5.3, другий - 1.3 8.3)

```
v1:2.3 5.3
v2:1.3 8.3
(1,-3)
```

Поліморфізм

Слово поліморфізм означає, що щось має багато форм. Як правило, поліморфізм виникає, коли існує ієрархія класів, і вони пов'язані успадкуванням.

Поліморфізм в C++ означає, що виклик методу призведе до виконання іншої функції в залежності від типу об'єкта, який викликає цей метод.
Розглянемо наступний приклад, коли базовий клас був отриманий іншими двома класами

```
#include <iostream>
using namespace std;
 class Shape {
   protected:
      int width, height;
   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

int main() {
   Shape *shape;
   Rectangle rec(10,7);
```

```
   Triangle  tri(10,5);

   // беремо адресу Rectangle
   shape = &rec;
   // отримуємо його площу
   shape->area();

   // беремо адресу Triangle
   shape = &tri;
      // отримуємо його площу
   shape->area();
}
```

Результат:

```
Parent class area :
Parent class area :
```

Причина неправильного виведення полягає в тому, що виклик функції area () встановлюється компілятором один раз як версія, визначена в базовому класі. Це називається статичним дозволом виклику функції, або *статичним зв'язком* - виклик функції фіксується до виконання програми. Це також іноді називається *раннім зв'язуванням*, оскільки функція area () встановлюється під час компіляції програми.

Але тепер давайте зробимо невелику модифікацію в нашій програмі і поставимо перед декларацією методу area () в класі Shape ключове слово virtual, щоб вона виглядала так:

```
class Shape {
   protected:
      int width, height;

   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      // чисто віртуальна функція pure virtual function
      virtual int area() = 0;
};
```

Код = 0 повідомляє компілятору, що функція не має тіла, а вище віртуальна функція буде називатися чисто віртуальною функцією.

```
Rectangle class area :70
Triangle class area :25
```

Поліморфізм (перевизначення) методів

Таким чином можна додавати нові дані і функції до класу через успадкування. Але як бути, якщо ми хочемо, щоб наш похідний клас успадкував метод від базового класу, але мати іншу реалізацію для нього? Саме тоді мова йде про поліморфізм, фундаментальне поняття в програмуванні ООП. Поліморфізм поділяється на два поняття: статичний поліморфізм і динамічний поліморфізм. Динамічний поліморфізм застосовується в C++, коли похідний клас замінює функцію, оголошену в базовому класі.

Ми реалізуємо цю концепцію, перевизначивши метод у похідному класі. Але для цього потрібно ввести поняття ***динамічного зв'язування***, ***статичного зв'язування*** і ***віртуальних методів***.

Припустимо, що ми маємо два класи, A і B. B є наслідником A і перевизначає реалізацію методу c (), що знаходиться в класі A. Тепер припустимо, що ми маємо об'єкт b класу B. Як інтерпретувати команду b.c ()?

Якщо b оголошений у стеку (не оголошений як вказівник або посилання), компілятор застосовує статичну прив'язку, це означає, що він інтерпретує (під час компіляції), що ми посилаємося на реалізацію c(), що знаходиться в B.

Однак, якщо ми оголошуємо b як вказівник або посилання класу A, компілятор не може знати, який метод викликати під час компіляції, оскільки b може бути типу A або B. Якщо це зроблено під час виконання, буде викликаний метод, який буде розташовано в B. Це називається **динамічним зв'язуванням**.

Якщо це буде вирішено під час компіляції, буде викликаний метод, який знаходиться в A. Це – **статичне зв’язування**.

Віртуальні методи

Віртуальні методи є відносно простими, але їх часто неправильно розуміють. Концепція є невід'ємною частиною розробки ієрархії класів щодо класів підкласів, оскільки вона визначає поведінку перевизначених методів у певних контекстах.

Віртуальними методами є функції членів класу, які ***можуть бути перевизначені в будь-якому класі***, ***що є нащадком того***, де вони були оголошені. Тіло методу потім замінюється новим набором реалізації в похідному класі.

Примітка 1: При перевизначенні віртуальних функцій можна змінювати приватний, захищений або відкритий стан доступу до стану методу похідного класу.

Розміщуючи ключове слово virtual перед декларацією методу, ми вказуємо, що коли компілятор має вирішити між застосуванням статичного зв'язування або динамічного зв'язування, він застосує динамічну зв'язування. В іншому випадку буде застосована статичне зв'язування.

Примітка 2: Хоча в нашому підкласі не потрібно використовувати ключове слово virtual (оскільки функція базового класу є віртуальною, всі заміщення підкласу з неї також будуть віртуальними), це хороший стиль для створення коду для майбутнього використання (для використання поза проекту).
Знову ж таки, це має бути зрозумілішим з прикладом:

```
#include <iostream>
class Base{
public:
  void method()  {
    std::cout << "Base::usual()" << std::endl;
  }
  virtual void virtual_method()  {
    std::cout << "Base::virtual()" << std::endl;
  }
};

class Son : public Base{
public:
  void method()  {
    std::cout << "Son::usual()" << std::endl;
  }
  virtual void virtual_method()  {
    std::cout << "Son::virtual()" << std::endl;
  }
};

int main(){
  Base base;
  Son son;

  Base *p_base = &son;
  Son *p_son = &son;
  //Son *p_wrong = &base; // Таке перетворення неможливо

  base.method(); // "Base::usual()"
  base.virtual_method(); // "Base::virtual()"

  son.method(); // "Son::usual()"
  son.virtual_method(); // "Son::virtual()"
  // Але тепер ….
  p_base->method();  // "Base::usual()"
  p_base->virtual_method();  // "Son::virtual()"!!!!
```

```
  p_son->method(); // "Son::usual()"
  p_son->virtual_method(); // "Son::virtual()"
}
```
Наші перші виклики method() і virtual_method() на двох об'єктах прості. Проте цікаві речі з нашим вказівником на змінну base, який є вказівником на тип Base. Оскільки метод method() не є віртуальним, то виклик до method() завжди буде викликати реалізацію, того класу в якому він викликається, тому отримаємо для p_base->method() та p_son->method() відповідно "Base::usual()" та "Son::usual()". А ось метод method пов'язану з типом вказівника, у цьому випадку реалізація береться з Son, тобто і для p_base->virtual_method() і для p_son->virtual_method() отримується однакове значення "Son::virtual()"!

Примітка: Пам'ятайте, що перевантаження та перевизначення є різними поняттями.

Виклики віртуальних функцій обчислювально дорожчі, ніж звичайні виклики функцій. Віртуальні функції використовують вказівники на вказівки від функцій і вимагають декількох додаткових інструкцій, ніж звичайні функції члена. Вони також вимагають, щоб конструктор будь-якого класу / структури, що містить віртуальні функції, ініціалізував таблицю вказівників на його віртуальні методи. Всі ці характеристики визначають компроміс між продуктивністю та дизайном. Слід уникати попереднього оголошення функцій віртуальними без існуючих структурних потреб. Майте на увазі, що віртуальні функції, які визначаються лише під час виконання, не можуть бути вбудованими (inline).

Примітка: Деякі потреби у використанні віртуальних функцій можна вирішити за допомогою шаблонів класу.

Чистий віртуальний метод

Є ще одна цікава можливість. Іноді ми взагалі не хочемо забезпечувати реалізацію нашої функції, але хочемо вимагати від людей, які роблять наслідник нашого класу, забезпечити реалізацію самостійно. Це називається **чистими віртуальними методами**.

Для позначення чистої віртуальної функції замість реалізації потрібно просто додати "= 0" після оголошення функції.

Абстрактні класи та віртуальні функції

Абстрактний клас, концептуально, клас, який не може бути створений і зазвичай реалізується як клас, який має одну або більше чистих віртуальних (абстрактних) функцій.

Чиста віртуальна функція є такою, яка повинна бути перевизначена будь-яким конкретним (тобто, не абстрактним) похідним класом. Це вказується в декларації з синтаксисом "= 0" у декларації функції-члена(метода).

```
class AbstractClass {
```

```
public:
  virtual void AbstractMemberFunction() = 0; /* чисто віртуальна
функція (pure virtual function) робить цей клас Abstract class.
*/
  virtual   void   NonAbstractMemberFunction1();   //   Virtual
function – віртуальна функція

  void NonAbstractMemberFunction2();
};
```

Взагалі абстрактний клас використовується для визначення реалізації і призначений для успадкування від конкретних класів. Це спосіб змусити *зробити контракт* між дизайном класу і користувачами цього класу. Якщо ми хочемо створити конкретний клас (клас, який може бути інстанційованим) з абстрактного класу, ми повинні оголосити і визначити відповідний метод класу для кожної абстрактної функції члена базового класу. В іншому випадку, якщо будь-яка функція-член базового класу залишається невизначеною, ми створимо новий абстрактний клас (іноді це може бути корисним).

Іноді ми використовуємо фразу "чистий абстрактний клас", що означає клас, який має виключно чисті віртуальні функції (і немає даних). Концепція інтерфейсу зпівставляється з чистими абстрактними класами в C ++, оскільки в C ++ не існує "інтерфейсної" конструкції так само, як, наприклад, в Java.

Покажемо приклад, в якому Vehicle є абстрактним базовим класом, оскільки він має абстрактний метод. Клас WheeledVehicle виводиться з базового класу. Він також містить дані, які є загальними для всіх колісних наземних транспортних засобів, а саме кількість коліс. Класи Car та Plane — наслідники відповідно WheeledVehicle та Vehicle. Наявність абстрактного класу дозвоялє працювати з усіма цими класамі одночасно:

```
#include <iostream>

class Vehicle {
 public:
     Vehicle() {}
     Vehicle(double v): speed(v) {}
     double TopSpeed() const {
        return speed;
     }

     virtual void print(std::ostream& ) const = 0;
 protected:
     int speed;
 };
```

```
class WheeledVehicle : public Vehicle {
 public:
     WheeledVehicle(){}
     WheeledVehicle(double v, unsigned wheels)
     : Vehicle(v), numberOfWheels(wheels) {}
     unsigned getNumberOfWheels() const {
       return numberOfWheels;
     }

     void  print( std::ostream& ) const{ // цей метод implicitly
virtual
       std::cout<< "Print Wheels " << numberOfWheels<<"\n";
    }
 protected:
     unsigned numberOfWheels;
};

class Car: public WheeledVehicle{
 public:
    Car(){}
    Car(double v, int vol)
    : WheeledVehicle(v,4), volume(vol)
    {}
    int engineVolume() const {
       return volume;
    }
    void  print( std::ostream& ) const{ // цей метод implicitly
virtual
       std::cout<< "Print Car, Wheels " << numberOfWheels<<"
vol:"<<volume<<"\n";
    }
  private:
    int volume;
  };

class Plane: public Vehicle {
  public:
     Plane(){}
     Plane(double v, double h)
     : Vehicle(v), maxHigh(h) {}
     double getMaxHigh() const {
       return maxHigh;
```

```
    }

    void  print( std::ostream& ) const{ // цей метод implicitly
virtual
            std::cout<<    "Print    Plane    ,maxHigh="    <<
maxHigh<<"\n";
    }
 private:
    double maxHigh;
};

int main(){
   Vehicle *mas[4] ;
   mas[0] = new WheeledVehicle(15.0,2);
   mas[1] = new Car(155.0,45);
   mas[2] = new Plane(855.0,15.0);
   mas[3] = new Car(125.0,55);

   for(int i=0;i<4;++i) {
     mas[i]->print(std::cout);
     delete mas[i];
  }
}
```

Результат:

```
Print Wheels 2
Print Car, Wheels 4 vol:45
Print Plane ,maxHigh=15
Print Car, Wheels 4 vol:55
```

Цей приклад показує, як можна поділитися подробицями реалізації серед ієрархії класів. Кожен клас додатково уточнює концепцію. В якості орієнтира, для зручності обслуговування і розуміння ви повинні спробувати обмежити наслідування не більш ніж на 3 рівнях. Найчастіше найкращим набором класів є чистий віртуальний абстрактний базовий клас для визначення загального інтерфейсу. Потім потрібно використовувати абстрактний клас для подальшого вдосконалення реалізації для набору конкретних класів і, нарешті, визначення набору конкретних класів.

Приклад використання чистого абстрактного класу. В представленому нижче прикладу навдена реалізація одного з класичних паттернів програмування, коли за допомогою спільного поліморфного віртуального методу передається інформація від різних класів у єдину точку (в даному випадку — кількість кутів передається в клас StoreAngles):

```
#include <iostream>

class StoreAngles{
   public:
   int angles;
   StoreAngles(){angles=0;}
};

class DrawableObject {
  public:
    virtual void angleCount(StoreAngles& ) const = 0; //рахує
кути
 };

 class Triangle : public DrawableObject {
 public:
   void angleCount(StoreAngles&) const; //малює трикутник
 };

 class Rectangle : public DrawableObject {
 public:
   void angleCount(StoreAngles&) const; //малює прямокутник
 };

 class Circle : public DrawableObject {
 public:
   void angleCount(StoreAngles&) const; //малює коло
 };


void Triangle::angleCount(StoreAngles& x) const{
    x.angles+=3;
}

void Rectangle::angleCount(StoreAngles& x) const{
    x.angles+=4;
}

void Circle::angleCount(StoreAngles& x) const{
    x.angles+=0;
}
```

```
int main(){

 DrawableObject* drawableList[3];

 StoreAngles drawingBoard;

 drawableList[0] = new Triangle();
 drawableList[1] = new Rectangle();
 drawableList[2] = new Circle();

 for(int i=0;i<3;++i) {
   DrawableObject *object = drawableList[i];
   object->angleCount(drawingBoard);
 }

 std::cout<<"total number of angles:"<<drawingBoard.angles;
}
```

Результат:

```
total number of angles:7
```

Майте на увазі, що об'єкти, які визиваються, не є повністю визначеними (без конструкторів або даних), але вони повинні дати загальне уявлення про силу визначення інтерфейсу. Після того, як об'єкти побудовані, код, який викликає інтерфейс, не знає жодної деталі реалізації викликаних об'єктів, тільки ті з інтерфейсу. Об'єкт StoreAngles є класом, який ніяк не повязаний з класами геометричних фігур, але він тепер може отримати інформацію від них та працювати з нею.

Зауважимо, що існує велика спокуса додати конкретні функції-члени і дані до чистих абстрактних базових класів. Якщо таке трапиться, в цілому це знак того, що інтерфейс недостатньо продуманий. Дані і конкретні функції-члени, як правило, мають на увазі конкретну реалізацію і як такі можуть успадковуватися від інтерфейсу, але не повинні бути цим інтерфейсом. Навпаки, якщо існує деяка спільність між конкретними класами, добре робити створення абстрактного класу, який успадковує його інтерфейс від чистого абстрактного класу і визначає загальні дані і функції-функції конкретних класів. Необхідно взяти певну обережність, щоб вирішити, чи слід використовувати спадкування або агрегацію. Занадто багато шарів успадкування можуть ускладнити обслуговування та використання класу. Як правило, максимально допустимі шари успадкування становлять близько трьох, якщо їх більше, зазвичай потрібен рефакторинг класів. Загальним тестом є "є" проти "має". Наприклад, на площі ***є*** прямокутник, але площа ***має*** набір сторін.

Віртуальний деструктор

Пам'ятайте, що якщо Foo використовує динамічно виділену пам'ять, ви повинні **визначити віртуальний деструктор ~ Foo () і зробити його віртуальним**, щоб подбати про звільнення об'єктів за допомогою вказівників до батьківських класів. Тому особливо важливо пам'ятати про визначення віртуального деструктора, навіть якщо він порожній у будь-якому базовому класі, оскільки помилка в ньому може створити проблеми з деструктором, створеним за замовчуванням, який не буде віртуальним.

Зверніть при цьому увагу, що кожен клас в ієрархії повинен мати власний деструктор, ала при приведенніі до нижчого поієрархії типу повинен спрацьовувати коректний деструктор, або інакше будемо мати проблеми з Memory Leak.

Чистий віртуальний деструктор

Кожен абстрактний клас повинен містити декларацію чистого віртуального деструктора.

Чисті віртуальні деструктори - це особливий випадок чистих віртуальних функцій (призначених для перевизначення у похідному класі). Вони завжди повинні бути визначені, і це визначення завжди має бути порожнім.

```
class Interface {
public:
  virtual ~Interface() = 0; //декларація чистого віртуального деструктора
};

Interface::~Interface(){} //визначення чистого віртуального деструктору (завжди повинно бути порожнім)
```

Коваріантні типи повернення

Коваріантні типи повернення - це можливість для віртуальної функції в похідному класі повернути вказівник або посилання на примірник самого себе, якщо версія методу в базовому класі робить це.

Приклад.

```
class base{
public:
  virtual base* create() const;
};

class derived : public base{
public:
  virtual derived* create() const;
};
```

Це дозволяє уникнути перетворення типів.
Примітка: Деякі старі компілятори не мають підтримки для коваріантних типів повернення. Для таких компіляторів існують обхідні шляхи.

Локальні(анонімні) класи

Локальний клас - це будь-який клас, який визначений у певному блоці оператора, в локальній області, наприклад, усередині функції. Це робиться так, як визначається будь-який інший клас, але локальні класи не можуть звертатися до нестатичних локальних змінних або використовуватись для визначення статичних членів даних. Ці типи класів корисні, особливо в шаблонних функціях, які будуть розглядатися пізніше.

```
void MyFunction(){
   class LocalClass
   {
   // ... визначення членів ...
   };
      // ... код, де використовується клас ...
}
```

4. Клас «рядок» (String)

Для роботи з рядками на C++ (починаючи з C++98) створений спеціальний клас string, який ще звуть клас роботи з рядком стандартної бібліотеки (STL).
Для порівняння роботи з рядками стандартних ANSI-рядків або рядків, що завершуються нульовим символом, та роботи з рядком класу string розглянемо роботу цих класів на прикладі:

Робота з ANSI-рядком	Робота з класом string

`#include <iostream>`
 	
`#include <cstring> // Required by strcpy()`
`#include <cstdlib> // Required by malloc()`
 	
`int main(int argc, char **argv)`
`{`
`    char CC[17];// C character string (16 characters + NULL termination)`
`    char *CC2;   // C character string. No storage allocated.`
 	
`    strcpy(CC,"This is a string");`
`    CC2 = (char *) malloc(17);       // Allocate memory for storage of string.`
	`#include <iostream>`
	`#include <string>`
	
	`int main(int argc, char **argv)`
	`{`
	`    std::string SS;    // C++ STL string`
	`    std::string SS2;   // C++ STL string`
	
	`    SS = "This is a string";`
	`    SS2 = SS;`
	
	`    std::cout << SS << "\n";`

```
    strcpy(CC2, CC);//strcpy(CC2,"This is a string");

    std::cout << CC << "\n";
    std::cout << CC2 << "\n";
}
```

```
    std::cout << SS2 << "\n";
}
```

Компіляція: g++ stringtest.cpp -o stringtest
Запуск: ./stringtest
Результат роботи обох програм повинен бути ідентичним:

```
This is a string
This is a string
```

Можна побачити, що робота з рядками в С-стилі та в стилі С++ є одночасно допустимими, але неважко побачити, що клас роботи з рядками С++ забезпечує більше функціональності та зручності. Рядок STL не вимагає попередньо виділеної пам'яті або виділення вручну, значно легше робиться присвоювання та ініціалізація таких рядків.
Клас рядків STL також надає багато корисних методів роботи з рядками.

Конструктори та методи класу string

Для створення, тобто декларації та ініціалізації рядків STL, очевидно потрібно підключити модуль реалізації цього класу #include <string> та викликати конструктор для цього класу.
Для класу string існують наступні варіанти конструкторів:

1. Конструктор по літералу:**:**
string <ім'я_змінної>(<Літерал>);
Приклад:

```
  string x("Literal");
    string sVar1("Програмування на Ci++");
```

2. Конструктор по ANSI-рядку (С-рядку):
char <ім'я_с_рядку>[] ; // С-рядок (Null terminated char)
*** /// ім'я_с_рядку ініціалізуєтсья
string <ім'я_змінної>(<ім'я_с_рядку>);
Приклад:

```
    char  cVar[10]= "Hello!";
    string sVar2(cVar);
```

3. Конструктор з декількох симовлів:
string <ім'я_змінної>(<кількість_символів>, <Символ>);

```
string sVar3(5, 'a');                // 5 символів 'a': "aaaaa"
```

4. Конструктор з іншого рядку та початкового символу:
string <ім'я_змінної>(<ім'я_іншого_рядку>, <початок_рядку>);

```
string Var1 ("String Constructor");
string sVar4(Var1, 7);  // Ініціалізує рядок з 8-го символу до кінця рядка  Var1: "Constructor"
```

5. Конструктор за допомогою початкового та кінцевого ітератору іншого рядку STL:
Конструктор за допомогою початкового та кінцевого ітератору іншого рядку STL:

```
string sVar5(Var1.begin(), Var1.end());
```

Крім того, звичайним чином визначений конструктор копіювання:
string <ім'я_змінної>(<ім'я_іншого_рядку>);

Аналогічно для цього класу існує звичайний деструктор:
~string();

Таблиця 4.1

Методи класу string

Функція/Операція	Опис
Var = *string2* Var.assign("*string-to-assign*")	Присвоєння string (operator=). При присвоюванні С "char*" типу, перевірте наявність NULL для запобагання прблемам (failure/crash). Наприклад, if(szVar) sVar.assign(szVar); де szVar змінна типу С "char *" та sVar типу "string".
Var.swap(*string2*) swap(*string1*,*string2*)	Метод міняє місцями поточний клас та string2. Функція swap міняє значення аргументів
Var += *string2* Var.append() Var.push_back()	Додає string/символи до кінця рядку(конкатинація). Приклад, string Var("abc"); виклик Var.append("xyz") створить рядок "abcxyz". Метод push_back() бере у якості аргменту лише символ (char)
Var.insert(size_t position, string) Var.insert(size_t position, char *) Var.insert(size_t position, string, size_t pos1, size_t len) Var.insert(size_t position, char *, size_t pos1, size_t len)	Вставляє рядок на дане місце position: вставляє перед даним місцем. Якщо він дорівнює 0, то вставляє перед рядком pos1: номер позиції першого символу рядку що вставляється len: довжина рядку, що вставляється
Var.erase() Var = ""	Очищує вміст рядку. аргументи не потрібні
+	Конкатинація

Функція/Операція	Опис
==, !=, <, <=, >, >=	Порівнює рядки
Var.compare(*string*) Var.compare(size_t pos1, size_t len, *string*) const; Var.compare(size_t pos1, size_t len1, const *string*, size_t pos2, size_t len2) const;	Порівнює рядки в С-стилі. Повертає int: • 0: якщо рівні. • -1: Не рівні. Перше неспівпадаюче значення в Var менше ніж у рядку, що порівнюється по ASCII таблці. • +1: Не рівні. Перше неспівпадаюче значення в Var більше ніж у рядку, що порівнюється по ASCII таблці. Тут *string* інший STL рядок або null terminated C рядок.
Var.length() Var.size() Var.capacity()	Повертає довжину пам'яті що зберігає рядок. Методи length(), size() та capacity() дозволяють отримати довжину рядку.
Var.max_size()	Повертає максимальний розмір рядку
Var.empty()	Повертає 1, якщо рядок порожній Повертає 0, якщо не порожній.
<<	Виводить в потік виводу
>> getline()	Вводить з потоку вводу
Var.c_str()	Конвертує рядок у С-рядок (що закінчується нулем). Не треба звільнювати цей результат!!!
Var.data()	Конвертує рядок у С-рядок, який **не закінчується** нулем! Не треба звільнювати цей результат!!!
Var[] Var.at(*integer*)	Квадратні дужки отримують доступ до елементів рядку. Повертає символ на даній позиції (починаючи з 0) . Приклад, Var("abc");тоді Var[2] == "c".
Var.copy(char *str,size_t len, size_t index)	str: виділений буфер для масиву char куди робиться копія len: кількість символів що потрібно зкопіювати index: стартова позиція в рядку (Var) звідки треба копіювати. Відлік починається з 0 Повертає кількість зкопійованих символів
Var.find(*string*) Var.find(*string, positionFirstChar*) Var.find(*string, positionFirstChar, len*)	Знаходить індекс першої появи підрядку в рядку. Повертає спеціальне беззнакове ціле. *positionFirstChar - звідки починати пошук в Var* len - довжина рядку який потрібно шукати (*string*) Якщо не знайдений підрядок повертає спецконстанту string::npos . Приклад if(Var.find("abc") == string::npos) cout << "Not found" << endl;
Var.rfind()	Знаходить індекс останньої появи підрядку в рядку.
Var.find_first_of(*string*, size_t position) Var.find_first_of(char *str, size_t position)	Находить рядки та підрядки тут *string* - STL string str - C string. Якщо position = 0, починає з початку рядку

Функція/Операція	Опис
Var.find_first_of(char *str, size_t position, size_t len)	
Var.find_last_of(*string*, size_t position) Var.find_last_of(char *str, size_t position) Var.find_last_of(char *str, size_t position, size_t len)	Находить рядки та підрядки з кінця. position: початок (тобто в даному рядку кінець рядку пошуку) len: кількість символів що треба знайти
Var.find_first_not_of() Var.find_last_not_of()	
Var.replace(pos1, len1, *string*) Var.replace(itterator1, itterator2, const *string*) Var.replace(pos1, len1, *string*, pos2, len2)	Замінює підрядок новими символами. pos2 та len2 потрібно лише якщо міняється лише підрядок *string.* *string* - інший STL string або C-рядок.
Var.substr(pos, len)	Повертає підрядок тексту від стартової позиції (pos) в рядку та довжині потрібного підрядку.
Var.begin() Var.end()	Ітератори
Var.rbegin() Var.rend()	Обернені ітератори

Типи ітераторів:

- string::traits_type
- string::value_type
- string::size_type
- string::difference_type
- string::reference
- string::const_reference
- string::pointer
- string::const_pointer
- string::iterator
- string::const_iterator
- string::reverse_iterator
- string::const_reverse_iterator
- string::npos

Інколи потрібно перевести з класу STL рядок в C-рядок. Для цього можна викликати метод c_str():

```
string sVar("GGGGG");
char* x  = svar.c_str();
```

Відповідно, для переводу STL рядку в числовий тип можна скористатись функціями переводу у число C-рядків:

```
string sVar1("LL1234");
int num = stol ( sVar1.c_str(), nullptr);
int num = atoi( sVar1.c_str() );
//if(sscanf(sVar1.c_str(), "%d", &i) != 1)
```

або починаючи з C++11:

```
int num =std::stoi( sVar1 )
```

Ще один варіант - використання функцій з модуля stringstream:

```
stringstream tmp(s);    // object from the class stringstream
int num = 0;
tmp >> num;
```

Для переводу числа в тип(клас) STL рядок починаючи з C++11 можна скористатись

```
res = to_string(number);
```

або скориставшись класом ostringstream з модулю sstream

```
#include <sstream> // ostringstream
string int2string (const int& number){
    ostringstream oss;
    oss << number;
    return oss.str();
}
string res =  int2string(number);
або функцією  sprintf :
   const size_t size=255;
   char text_num[size];
    sprintf(text_num, "%d", number); // Could have printed
directly using printf()
    string res1(text_num);
```

Приклади використання функцій рядку:

```
    string a("Preved Medved");
    string b{" i vsem privet"};
    string c;
    cout << a << " " << b << endl; // Output: abcd efg xyz ijk
    cout << "String empty: "<<boolalpha<<c.empty()<< endl;
    // String empty: 1  -  Yes it is empty. (TRUE)
    c = a + b; // concatenation
    cout << c << endl; // abcd efgxyz ijk
    cout <<"String length:"<<c.length()<< endl;
    // String length: 15
```

```
    cout << "String size: "<<c.size()<< endl;
    // String size: 15
    cout << "String capacity: "<< c.capacity() << endl;
    // String capacity: 15
    cout << "String empty: "<<c.empty()<<endl;
    // String empty: 0
    // Is string empty? No it is NOT empty. (FALSE)
    string d = c;
    cout << d << endl; // abcd efgxyz ij
    cout << "First character: " << c[0] << endl;// First
character: a
    // Strings start with index 0 just like C.
    string f("    Leading and trailing blanks      ");
    cout << "String f:" << f << endl;
    cout << "String length: " << f.length() << endl;
    // String length: 37
    cout << "String f:"<< f.append("ZZZ")<< endl;
    // String f:    Leading and trailing blanks      ZZZ
    cout << "String length: " << f.length() << endl;
    // String length: 40
    string g("abc abc abd abc");
    cout << "String g: " << g << endl;
    // String g: abc abc abd abc
    cout     <<     "Replace     12,1,\"xyz\",3:     "     <<
g.replace(12,1,"xyz",3) << endl;
    // Replace 12,1,"xyz",3: abc abc abd xyzbc
    cout << g.replace(0,3,"xyz",3) << endl;
    // xyz abc abd xyzbc
    cout << g.replace(4,3,"xyz",3) << endl;
    // xyz xyz abd xyzbc
    cout << g.replace(4,3,"ijk",1) << endl;
    // xyz i abd xyzbc
    cout << "Find: " << g.find("abd",1) << endl;
    // Find: 6
    cout << g.find("qrs",1) << endl;
    string h("abc abc abd abc");
    cout << "String h: " << h << endl;
    cout << "Find \"abc\",0: " << h.find("abc",0) << endl; //
Find "abc",0: 0
    cout << "Find \"abc\",1: " << h.find("abc",1) << endl; //
Find "abc",1: 4
```

```
    cout     <<     "Find_first_of     \"abc\",0:     "     <<
h.find_first_of("abc",0) << endl; // Find_first_of "abc",0: 0
    cout     <<     "Find_last_of     \"abc\",0:     "     <<
h.find_last_of("abc",0) << endl;
    // Find_last_of "abc",0: 0
    cout    <<    "Find_first_not_of    \"abc\",0:    "    <<
h.find_first_not_of("abc",0) << endl;
    // Find_first_not_of "abc",0: 3
    cout << "Find_first_not_of \" \": " << h.find_first_not_of("
") << endl;
    // Find_first_not_of " ": 0
    cout << "Substr 5,9: " << h.substr(5,9) << endl;
    // Substr 5,9: bc abd ab
    cout << "Compare 0,3,\"abc\": " << h.compare(0,3,"abc") <<
endl;
    // Compare 0,3,"abc": 0
    cout << "Compare 0,3,\"abd\": " << h.compare(0,3,"abd") <<
endl;
    // Compare 0,3,"abd": -1
    cout << h.assign("xyz",0,3) << endl;
    // xyz
    cout << "First character: " << h[0] << endl; // Strings
start with 0 // First character: x
```

Результат роботи:

```
Preved Medved  i vsem privet
String empty: true
Preved Medved i vsem privet
String length:27
String size: 27
String capacity: 30
String empty: false
Preved Medved i vsem privet
First character: P
String f:    Leading and trailing blanks
String length: 45
String f:    Leading and trailing blanks      ZZZ
String length: 48
String g: abc abc abd abc
Replace 12,1,"xyz",3: abc abc abd xyzbc
xyz abc abd xyzbc
xyz xyz abd xyzbc
```

```
xyz i abd xyzbc
Find: 6
18446744073709551615
String h: abc abc abd abc
Find "abc",0: 0
Find "abc",1: 4
Find_first_of "abc",0: 0
Find_last_of "abc",0: 0
Find_first_not_of "abc",0: 3
Find_first_not_of " ": 0
Substr 5,9: bc abd ab
Compare 0,3,"abc": 0
Compare 0,3,"abd": -1
xyz
First character: x
```

Приклад 2: Парсінг файлу

```
int parsing(const char* FILENAME){
    string::size_type posBeginIdx, posEndIdx;
    string::size_type ipos=0;
    string sLine, sValue;
    string sKeyWord;
    const string sDelim(" ");
    string sError;

    ifstream myInputFile(FILENAME, ios::in);
    if(!myInputFile){
        sError = "File could not be opened";
        return -1;// ERROR
    }
    cerr<<"Line";
    while(getline(myInputFile, sLine)){
             cerr<<sLine;
        if(myInputFile.bad()) break;
        if( !sLine.empty()){
            posEndIdx = sLine.find_first_of(sDelim);
            if(posEndIdx==string::npos) break;
            sKeyWord  =  sLine.substr(  ipos,  posEndIdx  );  //
Extract word
            posBeginIdx = posEndIdx + 1; // Beginning of next
word (after ':')
            cout<<posBeginIdx<<sKeyWord<<posEndIdx;
            ipos++;
```

```
            }
        }
    return ipos;
}
```

5. Виключення

Поняття виключень. Обробка виключень на C++. Ключові слова try/catch/throw. Особливості роботи виключень на C++. Створення власних виключень. Стандартні виключення C++.

Поняття виключень

Під час роботи програм зустрічаються ситуації які перешкоджають подальшої виконанню нормальної їх роботи. Наприклад, при діленні числа на нуль, програма завершить роботу не дивлячись на те, скільки користувач працював в програмі, і який обсяг даних було введено. Програма просто закриється.

І уявіть, якщо до цього користувач вносив в програму дані кілька годин і проводив розрахунки. При такому аварійному закритті програми всі ці дані і розрахунки пропадуть. Також може зустрітися така ситуація, коли програма намагається відкрити недоступний в цей момент файл, або запросити більше, ніж є пам'яті. В таких ситуаціях бажано вміти переривати небезпечний процес при цьому продовжуючи дію програми.

Такого роду ситуації програмістам треба намагатися передбачити і будувати програми так, щоб вони могли гнучко реагувати, а не аварійно закриватися.

Відоме таке визначення винятків в C++:

В мові C++ виключення - це реакція на нештатну ситуацію, що виникла під час виконання програми, наприклад при виділенні пам'яті або при відкритті файлу Виключення дозволяють передати управління програмою з однієї частини в іншу.

Розглянемо механізм роботи винятків в C++ на простому прикладі. У ньому ми передбачаємо той випадок, що в якийсь момент під час розрахунків програми, може зустрітися ділення числа на нуль. Наберіть і зкомпілюйте код, записаний нижче. Щоб переконатися в тому, як реагує програма на таку ситуацію, присвойте 0 в змінну **n** (вона виступає дільником в прикладі).

Приклад 1:

```
#include <iostream>
#include <clocale>

int main(){
    setlocale(LC_ALL, "ukr");
    int n;
```

```
    for(int i=0;i<3;i++){
        std::cout<<"n=";
        std::cin >> n;
        std::cout << "n*2 = " << 2 * n << "\n";
        std::cout << "1/n = " << 1 / n << "\n";
        std::cout << "1+n=" << 1 + n << "\n";
        std::cout  <<  "================================="  <<
"\n";
    }
    std::cout<<"Програма виконала 3 ітерації циклу";
}
```

Цикл **while** повинен відпрацювати три рази. Дійсно, якщо ми будемо вводити звичайні дані, то результат буде наступний:

```
n=1
n*2 = 2
1/n = 1
1+n=2
=================================
n=2
n*2 = 4
1/n = 0
1+n=3
=================================
n=3
n*2 = 6
1/n = 0
1+n=4
=================================
Програма виконала 3 ітерації циклу
```

Але якщо ввести значення 0 в змінну **n**, програма не пройде до кінця другий крок циклу, вона перерветься.

```
n=1
n*2 = 2
1/n = 1
1+n=2
=================================
n=0
n*2 = 0
Floating point exception
```

Отже, програма не завершила роботу.

У наступному лістингу, ми виправимо це упущення – добавимо в програму декілька компонентів, які допоможуть зреагувати на цю ситуацію без переривання роботи програми. А саме:

- блок **try** или **try**-блок (спроба);
- генератор виключення– блок **throw**(обробити, запустити);
- обробник виключення, який робить перехоплення виключення-команда **catch** (зловити, ловити).

Виключення працює наступним чином. Програміст прописує в коді (в **try**-блоці) конкретну умова, що якщо змінна **num2** буде дорівнювати 0, то в такому випадку необхідно генерувати виняток в **throw**. Далі, те що було згенеровано в **throw**, перехоплює **try**-блок (у вигляді параметра функції) і програма виконає той код, який прописаний в цьому блоці.

Приклад 2:

```
#include <iostream>
#include <clocale>

int main(){
    setlocale(LC_ALL, "ukr");
    int n;

    for(int i=0;i<3;i++){
        std::cout<<"n=";
        std::cin >> n;
        std::cout << "n*2 = " << 2 * n << "\n";

        try { // тут код, який може викликати помилку
            if (n == 0) {
                throw 42; // генерувати ціле число 42
            }
           std::cout << "1/n = " << 1 / n << "\n";
        }
        catch (int thr) { // сюди передається число, яке згенерував throw
            std::cout << "Помилка №" << thr << " - ділення на 0!!!" << '\n';
        }
        std::cout << "1+n=" << 1 + n << "\n";
        std::cout  <<  "================================"  << "\n";
    }
```

```
    std::cout<<"Програма виконала 3 ітерації циклу";
}
```

Результат:

```
n=1
n*2 = 2
1/n = 1
1+n=2
================================
n=0
n*2 = 0
Помилка №42 - ділення на 0!!!
1+n=1
================================
n=2
n*2 = 4
1/n = 0
1+n=3
================================
Програма виконала 3 ітерації циклу
```

На перший погляд, може здатися, що насправді простіше було б цю помилку обробити в стилі C — просто додати умову if (n == 0) з відповідним кодом. Але обмеження цього підходу, що ми зобов'язані обробити цей код безпосередньо в тому місці, де ми написали цю умову, в той час як за допомогою виключень ми можемо обробити її зокрема і в функції, яка викликає ту функцію, де сталося помилка. Крім того, використання виключень дозволить робити обробку помилок в більш об'єктно-орієнтовному стилі.

Синтаксис блоку виключення

Винятки забезпечують спосіб реагування на виняткові обставини (наприклад, помилки під час виконання) в програмах шляхом передачі керування до спеціальних функцій, які називаються обробниками.

Щоб зловити винятки, частина коду поміщається під інспекцію виключення. Це робиться шляхом укладання тієї частини коду в try-block. Коли виникає виняткова обставина в межах цього блоку, вилучається виняток, який передає керування обробнику винятку. Якщо не виключено жодного виключення, код продовжується нормально, і всі обробники ігноруються.

Виняток виникає за допомогою ключового слова throw з блоку try. Обробники винятку оголошуються ключовим словом catch, який має бути розміщений відразу після блоку try:

Приклад 2:

```
// Виключення та його обробка
```

```
#include <iostream>

int main () {
  try  {
    throw 20;
  }
  catch (int e)  {
    std::cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
}
```

Результат:

```
An exception occurred. Exception Nr. 20
```

Код під обробкою виключень укладений у блок спроби. У цьому прикладі цей код просто викидає виняток:

```
throw 20;
```

Вираз throw приймає один параметр (в даному випадку це ціле значення 20), який передається як аргумент обробнику винятку.

Обробник винятку оголошується з ключовим словом catch відразу після закриття фігурної дужки блоку try. Синтаксис catch схожий на звичайну функцію з одним параметром. Тип цього параметра дуже важливий, оскільки тип аргументу, переданий виразом throw, перевіряється проти нього, і тільки в тому випадку, якщо вони збігаються, виняток виявляється цим обробником.

Кілька обробників (тобто, вирази уловлювання) можуть бути прив'язані; кожен з іншим типом параметрів. Виконується тільки обробник, тип аргументу якого відповідає типу винятку, зазначеного у операторі throw.

Якщо в якості параметра catch використовується трикрапка (еліпсис) (...), то обробник відловлюватиме будь-яке виключення, незалежно від типу викинутого виключення. Це можна використовувати як обробник за умовчанням, який ловить усі винятки, які не виявлено іншими обробниками:

```
try {
  // код
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

У цьому випадку, останній обробник вловлюватиме будь-яке виключення типу, що не є ані int, ані char.

Після того, як виняток було оброблено програмою, виконання продовжується після блоку try-catch, а не після оператора throw !

Також можна вкласти блоки try-catch в більш зовнішні блоки try. У цих випадках ми маємо можливість, що внутрішній блок лову переадресовує виняток на свій зовнішній рівень. Це робиться за допомогою виразів throw без аргументів. Наприклад:

```
try {
```

```
  try {
      // тут деякий код
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Більш реалістичний код використання виключень. Нехай в нас на вхід подається бінарний файл, який повинен бути записаний в деякому форматі, зокрема на його початку буде записано перше число, яке дорівнює 42. Якщо ми бажаємо написати гарну програму нам потрібно обробити наступні ситуації:

- файл з даним іменем не був знайдений;
- ми не змогли з файлу нічого прочитати;
- перш число не дорівнює нашому специфікатору 42;
- тощо.

Ці ситуації можна обробити за допомогою виключень.

Приклад.

```
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
#include <exception>

int file_processing(std::string full_path){
   std::fstream file(full_path.c_str(), std::ios::binary);
   if(file.is_open()) {
        int magic_number = 0;

        int    res    =    file.read((char    *)&magic_number,
sizeof(magic_number));
        if(res!=4){
           throw "error";
        }

        if(magic_number!=42) throw magic_number;
        // some code
    } else {
```

```
        throw std::runtime_error("Unable to open file `" +
full_path + "`!");
    }

return 0;
}

int main(int argc, char **argv){
    try{
        file_processing("1.txt");
    }
    catch(std::runtime_error e){
        std::cout<<"error: "<<e.what()<<"\n";
    }
    catch(int a){
        std::cout<<"a="<<a<<" is not magic!\n";
    }
    catch(const char* a){
        std::cout<<"can't read 1 number!\n";
    }
}
```

Можливі результати (немає файлу, файл порожній, магічне число не рівне 42):

```
Unable to open file `1.txt`!

can't read 1 number!

a=2609 is not magic!
```

Примітка: Старий код може містити динамічні специфікації виключень. Вони тепер застаріли в C++, але досі підтримуються. Характеристика динамічного виключення слідує за декларуванням функції, додаючи до неї специфікацію. Наприклад:

double myfunction (*char* param) *throw* (*int*);

Це оголошує функцію myfunction, яка приймає один аргумент типу char і повертає значення типу double. Якщо ця функція кидає виняток іншого типу, відмінного від int, функція викликає std :: unexpected замість пошуку обробника або виклику std :: terminate.

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

```
    // дана функція може згенерувати лише int, char и double
(або нічого)
    void Xhandler(int test) throw (int, char, double){
    if(test==0) throw test; // генерація int
    if(test==1) throw 'a'; // генерація char
    if(test==2) throw 123.23; // генерація double
    }
    int main(){
    cout << "start\n";
    try {
    Xhandler(1); // спроба передати 1 та 2 в Xhandler()
    }
    catch (int i) {
    cout << "Caught an integer\n";
    }
    catch (char c) {
    cout << "Caught char\n";
    }
    catch(double d) {
    cout << "Caught double\n";
    }
    cout << "end";
    return 0;
    }
    /*
    // дана функція не може згенерувати виключення
    void Xhandler(int test) throw(){
    //
    //if(test==0) throw test;
    //if(test==1) throw 'a';*/
    //if(test==2) throw 123.23;
    }
    */
```

Якщо цей специфікатор *throw* залишиться порожнім без типу, це означає, що std :: unexpected викликається для будь-якого виключення. Функції без специфікатора виключення (регулярні функції) ніколи не називають std :: unexpected, але дотримуються звичайного шляху пошуку свого обробника виключень.

Тобто можливі наступні варіанти:

- *int* myfunction (*int* param) *throw*(); *//Не очікуються жодні виключення*
- *int* myfunction (*int* param); *// звичайна обробка виключень*

• <new> std :: nothrow // не очікується виключення

extern const nothrow_t nothrow;

Це значення константи використовується як аргумент для оператора new та new[] для вказівки, що ці функції не повинні викидати виняток при відмові, а замість цього повертають нульовий вказівник.

За замовчуванням, коли новий оператор використовується, щоб спробувати виділити пам'ять, а функція обробки не може зробити це, викидається помилка bad_alloc. Але коли nothrow використовується як аргумент для new, замість нього повертається нульовий вказівник.

Ця константа (nothrow) є лише значенням типу nothrow_t, з єдиною метою запуску перевантаженої версії оператора функції new (або оператора new []), що приймає аргумент цього типу.

У C++ оператор нової функції може бути перевантажений, щоб взяти більше одного параметра: перший параметр, переданий оператору новою функцією, завжди є розміром пам'яті, яку слід виділити, але додаткові аргументи можуть бути передані цій функції, уклавши їх в дужки в новому виразі. Наприклад:

int * p = new (x) int;

є коректним виразом, який у певний момент викликає оператор

new(sizeof (int), x).

За замовчуванням одна з версій оператора new перевантажена, щоб прийняти параметр типу nothrow_t (як nothrow). Саме значення не використовується, але ця версія оператора new повинна повернути нульовий вказівник у разі помилки замість того, щоб кидати виняток.

Те ж саме стосується new оператора [] і оператора функції new [].

Приклад:

```
// nothrow example
#include <iostream>     // std::cout
#include <new>          // std::nothrow
int main () {
  std::cout << "Attempting to allocate 1 MiB... ";
  char* p = new (std::nothrow) char [1048576];

  if (!p) {                 // null вказівники implicitly
переводяться до false
    std::cout << "Failed!\n";
  }
  else {
    std::cout << "Succeeded!\n";
    delete[] p;
  }
}
```

Проблема обробки виключення в С -коді.

Суттєвою проблемою застосувань виключень в С++ є обернена сумісність з С-кодом, шо призводить до того, що він не генерує виключень, а отже його неможливо піймати:

```
#include <iostream>
#include <cstring>
#include <string>

int main(){
    int a = 25, b = 5, c = 0;
    std::cout << "a=";
    std::cin >> a;
    std::cout << "b=";
    std::cin >> b;

    try  {
        c = a / b;
    }
    catch (...)    {
        std::cout << "division by zero";
    }
```

Запустіть цей код. Програма буде помилково завершуватись якщо b = 0. Наступний код може вирішити цю проблему:

```
#include <iostream>
#include <cstring>
#include <string>

int main(){
    int a = 25, b = 5, c = 0;
    std::cout << "a=";
    std::cin >> a;
    std::cout << "b=";
    std::cin >> b;

    try {
        if (b == 0)   throw "division by zero";
        c = a / b;
    }
    catch (const char* e)
    {
        std::cout << e << std::endl;
    }
    std::cout << "Result is " << c << std::endl;
}
```

або більш функціональним шляхом:

```
#include <iostream>
#include <cstring>
#include <string>

int dilennya(int n1, int n2){
    if (n2 == 0) {
        throw 42;
    }
    return n1 / n2;
};

int main(){
    int a = 25, b = 5, c = 0;
    std::cout << "a=";
    std::cin >> a;
    std::cout << "b=";
    std::cin >> b;

    try {
       c = dilennya(a,b);
    }
    catch (int x) {
        std::cout << "division by zero\n";
    }
    std::cout << "Result is " << c << std::endl;
}
```

Повторна генерація виключення

Іноді виникає потреба у тому щоб виключення було оброблено не в тій функції F, в якої воно було викликано, а в функції G в який викликалась функція F, чи навіть можливо, в якійсь з наступний функції з ієрархії викликів. Це можливо зробити за допомоги повторної генерації виключень.

Приклад. Повторна генерація виключення

```
#include <iostream>

void Xhandler(){
  try {
    throw "first"; // генерация char *
    }
   //помилка 1
  catch (const char*) { // перехват char *
   std::cout << "Caught 1st char * exception inside
Xhandler\n";
```

```
        throw "second"; // повторна генерація char * зовні
функції
      }
    }
    int main(){
     std::cout << "Start\n";
      try{
         Xhandler();
       }
      //помилка 2
      catch(const char *) {
        std::cout  <<  "Caught  2nd  char  *  exception  inside
main\n";
       }
      std::cout << "End\n";
    }
```

Результат:

```
Start
Caught 1st char * exception inside Xhandler
Caught 2nd char * exception inside main
End
```

Клас std::exception і створення власного виключення

Всі об'єкти, що є виключеннями від компонент стандартної бібліотеки, виводяться з цього класу. Таким чином, всі стандартні винятки можуть бути виявлені шляхом лову цього типу за посиланням.

Інтерфейс класу exception:

```
class exception {
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
}
```

Тобто ми маємо наступні функції-члени:

- конструктор виключення (публічний метод);
- оператор присвоєння - копіює виключення (публічний метод);
- метод , шо повертає рядок, який ідентифікує виключення (публічний віртуальний метод);

- деструктор (знищувач) виключення (публічний віртуальний метод).

Є можливість створити власне виключення шляхом наслідування та перевантаження функціональності класу exception. У вказаному прикладі, використовується клас std::exception для створення власного виключення:

Приклад.

```
#include <iostream>
#include <exception>
using namespace std;

struct MyException : public exception {
   const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      // Дії при інших помилках
   }
}
```

Результат:

```
MyException caught
C++ Exception
```

Тут what() - публічний метод, що створений в класі виключення і він перевантажується класами нащадками. Він повертає причину виключення як рядок.

В наступному прикладі ми перевантажили власне виключення для того, щоб виводити рядок в якому міститься помилка.

Приклад:

```
#include <iostream>
#include <exception>
struct MyException : public std::exception {

   int line;
   MyException(int m){
```

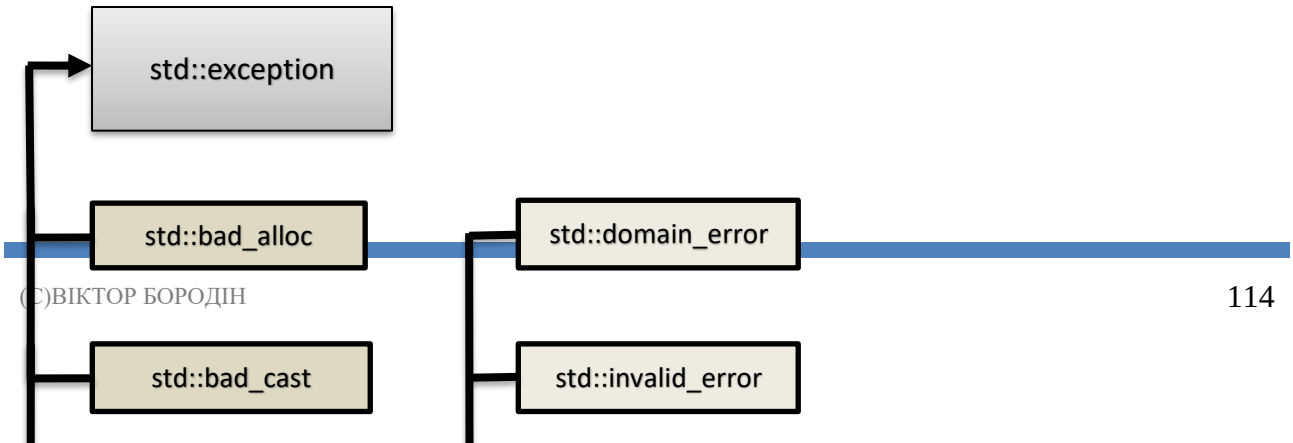
```
            line = m;
         }
      const char * what () const throw () {
         return "My C++ Exception";
      }
      int getLine(){
          return line;
     }
};

int my_function() {
   try {
      throw MyException(2);
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << "\t";
      std::cout << "on line: "<< e.getLine() << std::endl;
   } catch(std::exception& e) {
      //Other errors
      std::cout << e.what() << std::endl;
   }
}

int main(int argc, char **argv){
 my_function();
}
```

Результат:

```
MyException caught
C++ Exception  on line: 2
```

Стандартні виключення C++ (Standard Exceptions)

C++ містить деякі стандартні виключення, що містяться в заголовному файлі **<exception>** який можна підключати в програми. Ці класи-виключення підпорядковуються ієрархії класів, яка зображена нижче:



В таблиц 5.1і містяться описи даних виключень.

Таблиця 5.1

Опис стандартних виключень

Виключення (Exception) та його опис
std::exception Виключення — батьківський клас для віх стандартних Сі++ виключень.
std::bad_alloc Кидається при не виконанні **new**.
std::bad_cast викидається при поганому **dynamic_cast**.
std::bad_exception Корисне виключення, що викидається при виключенні, що не оброблюється С++ програмою.
std::bad_typeid Виключення, що викидається **typeid**.
std::logic_error виключення про логічну помилку, що теоретично може бути визначено при читанні коду

std::domain_error
виключення, що позначає, що математичні оператор застосовується в некоректній області визначення

std::invalid_argument
Викидається при направильних аргументах функції

std::length_error
Занадто вилике std::string створюється.

std::out_of_range
Виключення що викидається методом 'at', наприклоад в std::vector та std::bitset<>::operator[]().

std::runtime_error
виключення про логічну помилку, що теоретично не може бути визначено при читанні коду.

std::overflow_error
Переповнення типу при математичній операції

std::range_error
Виключення при роботі зі змінної з неправильного діапазону значень

std::underflow_error
Недостаньої повний математичний тип.

Ці виключення можна відловлювати якщо використовуються чисто C++ (не C!!!) класи та функції, такі як string, vector, static_cast тощо.

Приклад 1:

```
#include <iostream>
#include <new>

int main(){
//memory bad_alloc processing
    int * arr;
    try {
        arr = new int[1000000000];
        std::cout<<"Memory allocated \n";
         delete[] arr;
    } catch(std::bad_alloc ex) {
        std::cout<<"bad allocation "<<ex.what();
    }
}
```

Приклад 2:

```
#include <iostream>
#include <vector>
#include <exception>

int main(){
```

```
// out of range processing
//int x[9];
std::vector<int> x(9);
    try {
        for(int i =0; i<=10; ++i) {
            //x[i]
            x.at(i)=i; //
        }
        std::cout<<"Все гаразд"<<std::endl;
    } catch(std::out_of_range exception) {
        std::cout<<"Out of range";
        std::cout<<exception.what();
    } catch(...) {
        std::cout<<"невідома проблема";
    }
}
```

Приклад 3:

```
// Приклад відловлення bad_typeid  exception
#include <iostream>       // std::cerr
#include <typeinfo>       // operator typeid
#include <exception>      // std::exception

class Polymorphic {
virtual void member(){}
};
int main () {
  try  {
    Polymorphic * pb = 0;
    typeid(*pb);  // Буде bad_typeid exception
  }
  catch (std::exception& e)  {
    std::cerr << "exception caught: " << e.what() << '\n';
  }
  return 0;
}
```

Також на базі цих стандартних класів можна зробити власний клас виключення

Приклад (Створення власного виключення):

```
#include <iostream>
// Створюємо власну чергу та виключення для нього
class BasicQueueException : public std::logic_error {
```

```
    public:
     explicit          BasicQueueException(const          char*
message):std::logic_error(message) {}

    };
    // наслідування від вже побудованого виключення
    class EmptyQueueException : public BasicQueueException {
    public:
     explicit  EmptyQueueException():BasicQueueException("Queue
is empty") {}
     const char* what (){
          return "EmptyQueueException";
       }
    };

    class Queue{

     public:
      bool isEmpty(){ //stub
        return true;
      }
      void pop();
    };

    // в реалізації черги викликаємо виключення
    void Queue::pop() {
      if(isEmpty()) throw EmptyQueueException();
    }
    // ... Десь в коді можна відловити це виключення
    int main(){
        Queue q;
    try{
      q.pop();
    }
    catch(BasicQueueException& e) {
      std::cerr << "error "<< e.what() << std::endl;
    }
    }
```

6. Перетворення типів

Перетворення типів та їх види на C++. Неявні перетворення простих типів та класів.

Керування перетворенням класів. Ключове слово explicit.
Оператори перетворення типів. Динамічне визначення типів. Оператор typeid.
Явні перетворення типів: static_cast, dynamic_cast, const_cast, reinterpret_cast.

Неявні перетворення

В достатньо багатьох програмах на типізованих мовах програмування зустрічаються ситуації, коли треба поміняти тип змінної або присвоїти значення одного типу змінній іншого типу. Для коректного такого перетворення потрібно зробити перетворення типу. Ці перетворення можуть бути явними, тобто явно вказаними в програмному коді, або неявними, тобто такими, що здійснюються компілятором автоматично, якщо він бачить потребу в цьому.

Неявні перетворення на C/C++ здійснюються автоматично, коли значення копіюється в сумісний тип. Наприклад:

```
short = 2000;
int y;
y= x;
```

Тут тип значення змінної x підвищується від типу short до типу int без необхідності явного оператора перетворення. Це відомо як стандартне **(**або вроджене, implicit**)** перетворення. Стандартні перетворення відбуваються з основним типами даних і дозволяють перетворення типу між числовими типами (short до int, int до float, double до int тощо), а також можуть проводити перетворення з bool та робити деякі перетворення вказівників.

Зокрема воно може робити перетворення в int з якогось меншого цілого типу, або в подвійне(double) з float, відоме як просування, і гарантує, що воно буде точно таким же значенням у типі призначення. Інші перетворення між арифметичними типами не завжди можуть точно представляти одне значення, зокрема:

- Якщо негативне ціле значення перетворюється в беззнаковий тип, то результуюче значення відповідає побітовий зміні його першого біту до 1 (тобто -1 стає найбільшим значенням, що представляється типом, -2 другим за величиною, ...).
- Перетворення з / до типу bool вважають рівним false значення, що еквівалентні нулю (для числових типів) та нульовому вказівником (для типу вказівника), а true еквівалентно всім іншим значенням і перетворюється в еквівалент 1.
- Якщо перетворення відбувається від типу з плаваючою крапкою (дійсного) до цілочисельного типу, значення обрізається (десяткова частина видаляється). Якщо результат лежить за межами діапазону репрезентативних значень за типом, перетворення викликає невизначену поведінку.

•В іншому випадку, якщо перетворення відбувається між числовими типами одного і того ж виду (цілочисельне або ціле число або з плаваючою крапкою – до типу з плаваючою крапкою), перетворення є коректним, але значення є специфічним для реалізації (і може бути не кросплатформеним).

Деякі з цих перетворень можуть означати втрату точності, про що компілятор може сигналізувати попередженням. Цього попередження можна уникнути за допомогою явного перетворення.

Для не фундаментальних типів масиви та функції неявно перетворюються на вказівники, а вказівники взагалі дозволяють проводити такі перетворення:

• нульові вказівники можуть бути перетворені до вказівників будь-якого типу;

• вказівники на будь-який тип можуть бути перетворені в вказівники на void;

• перетворення вказівників: вказівники на похідний клас можуть бути перетворені на вказівник доступного та однозначного базового класу, не змінюючи його const або volatile специфікації.

Приклади:

```
1) short a=200000000; // warning: overflow in implicit constant conversion
int b;
b = a; // це перетворення неявне та цілком законне a=-15872,b=-15872

2) int a1=200000000; // все ОК
short b1;
b1 = a1; // а ось це перетворення підозріле a1=200000000, але b1=-15872

3) unsigned x=-1; // так, але це буде 4294967295, а не -1
int y;
y = x; // а тут буде знов -1

4) int x1=-1; //Ок
unsigned y1;
y1 = x1; // нам потрібно 4294967295?

5) int x2=2000000; //ok
float y2; double z2;
y2 =x2;  // ok
z2 =y2; //ok

6) double x3=2000000.12344;
```

```
float y3; int z3;
y3 = x3; // обрізка точності..
z3 = x3; // ціла частина, але чи саме вона нам потрібна?
```

Керування перетворенням типів

Ми розглянули автоматичне перетворення типів (неявне перетворення) і згадували, що деякі з них можуть бути визначені користувачем.

Визначене користувачем перетворення з класу в інший клас можна виконати, надавши конструктор у цільовому класі, який приймає клас джерела як аргумент, Target (const Source & a_Class) або надаючи цільовий клас оператором перетворення, як оператор Source().

Іноді потрібно запобігати копіюванню класів. Це потрібно, наприклад, для запобігання проблем, пов'язаних з пам'яттю, які призведуть до того, що конструктор копіювання за замовчуванням або оператор присвоєння за замовчуванням ненавмисно застосований до класу C, який використовує динамічно виділену пам'ять, де конструктор копіювання та оператор присвоєння, ймовірно, перевищують наявні ресурси.

Деякі вказівки щодо стилю пропонують, щоб усі класи не могли копіювати за замовчуванням, а лише конструктором копіювання, якщо це має сенс. Інші (погані) рекомендації говорять про те, що ви завжди повинні чітко писати конструктор копіювання та оператор призначення копій; це насправді погана ідея, оскільки вона додає до зусиль з технічного обслуговування, додає до роботи для читання класу, робить більше помилок, ніж при використанні неявно оголошених конструкторів, і не має сенсу для більшості типів об'єктів. Розумним керівництвом є думати про те, чи копіювання має сенс для типу; якщо це так, то спочатку спробуйте організувати, щоб операції копіювання, створені компілятором, працювали правильно (наприклад, утримуючи всі ресурси за допомогою класів керування ресурсами, а не через вказівники та посилання), а якщо це не розумно, дотримуйтесь закону трьох. Якщо копіювання не має сенсу, його можна заборонити як показано нижче.

Просто задекларуйте конструктор копіювання і оператор присвоювання, і зробить їх приватними. Не визначайте їх. Оскільки вони не є захищеними або публічними, вони недоступні за межами класу. Використання їх у класі дасть помилку лінкера, оскільки вони не визначені.

```
class C
{
  ...
   private:
    // Not defined anywhere
    C (const C&);
    C& operator= (const C&);
};
```

Пам'ятайте, що якщо клас використовує динамічно виділену пам'ять для членів даних, необхідно визначити процедури випуску пам'яті в деструкторі ~ C (), щоб звільнити виділену пам'ять.

Клас, який оголошує лише ці дві функції, можна використовувати як приватний базовий клас, так що всі класи, які приватно успадковують такий клас, заборонять копіювання.

Неявні перетворення з класами

У світі класів неявні перетворення можна керувати за допомогою трьох методів

(функцій-членів класу):
• **Конструктори з одним аргументом:** дозволяють неявне перетворення з певного типу для ініціалізації об'єкту.
• **Оператор присвоєння**: дозволяє непряме перетворення з певного типу до заданого типу.
• **Оператор-перетворення типів:** дозволяє непряме перетворення до певного типу.

Приклад (перетворення класу за допомогою конструктору, оператору присвоєння та оператору перетворення типів):

```
///Перетворення класів:  implicit conversion of classes:
#include <iostream>
#include <ostream>

class A { // перший клас
   friend std::ostream& operator<< (std::ostream& out, const
A& a){
      out<< "print A\n";
      return out;
   }
};

class B { // другий клас
public:
  //  Переторення  у  констуркторі:    conversion  from  A
(constructor):
  B (const A& x) {
       std::clog<<"construct B(A):\n";
  }
  //Перетворення   у   присвоєнні:   conversion   from   A
(assignment):
  B& operator= (const A& x) {
       std::clog<<"assign B=A:\n";
  return *this;}
  /*Перетворення оператором перетворення:   conversion to A
(type-cast operator) */
  operator A() {
    std::clog<<"conversion A(B):\n";
    return A();
   } // ми створили власний оператор переторення

  friend std::ostream& operator<< (std::ostream& out, const
B& a){
```

```
        out<< "print B\n";
        return out;
    }

};

int main (){
  A foo;
  B bar = foo;    // викликає конструктор B(A)
  std::cout<<foo<<bar;
  bar = foo;      // викликає присвоєння B=A
  std::cout<<bar;
  foo = bar;      // викликає перетворення типу A(B)
  std::cout<<foo;

}
```

Результат:

```
construct B(A):
print A
print B
assign B=A:
print B
conversion A(B):
print A
```

Оператор перетворення типу (type-cast operator) використовує певний синтаксис: він використовує ключове слово оператора, за яким слід тип призначення та порожній набір дужок. Зауважте, що тип повернення - тип призначення, і тому він не вказується перед ключовим словом оператора.

Ключове слово explicit

Під час виклику функції C ++ дозволяє здійснювати одне неявне перетворення для кожного аргументу. Це може бути дещо проблематичним для занять, адже це не завжди те, що призначено. Наприклад, якщо до останнього прикладу додати таку функцію:

```
void fn (B arg) {}
```

Ця функція бере аргумент типу B, але її також можна викликати з об’єктом типу A як аргумент:

```
  bar.fn(bar);
```

```
  bar.fn(foo);
```

Результат:

```
use fn(B):
print B

construct B(A):
use fn(B):
print B
```

Це може бути як тим що потрібно, так і не тим, що потрібно. Але, у будь-якому випадку, цьому можна запобігти, позначивши даний конструктор ключовим словом *explicit*:

```
// explicit:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  explicit B (const A& x) {}
  B& operator= (const A& x) {return *this;}
  operator A() {return A();}
};

void fn(B x) {}

int main (){
  A foo;
  B bar (foo);
  bar = foo;
  foo = bar;

  //fn (foo);   // not allowed for explicit ctor – не дозволено при explicit
  fn (bar);
}
```

Крім того, конструктори, позначені *explicit*, не можуть бути викликані синтаксисом, подібним до присвоєння.

У наведеному вище прикладі bar не міг бути побудований так:

```
B bar = foo;
```

Методу також можна вказати як *explicit*. Це запобігає неявним перетворенням так само, як це роблять чітко визначені конструктори для типу призначення.

Явні перетворення типів

C++ - мова сильного типу. Багато перетворень типів, особливо ті, що передбачають різну інтерпретацію значення, вимагають явного перетворення, відомого в C++ як кастинг типів. В C++ існує два основні синтаксиси для загального кастингу типу: функціональний та C-подібний:

```
double x = 10.3;
int y;
y = int (x);    //функціональний ( functional notation)
y = (int) x;    // C-подібний (c-like cast notation)
```

Приклад:

```
int a=2000;
short b;
unsigned c;
b=(short)a;    /*   варіант b=a -   теж можливий але з перетворенням ми чітко вказуємо, що розуміємо що робимо */
c = unsigned(a); // функціональний варіант
float d = (float)a;
```

Функціональність цих загальних форм перетворень типів достатня для більшості потреб роботи з основними типами даних. Однак ці оператори можуть бути застосовані до класів та і до вказівників на класи, що може призвести до синтаксично правильний коду, що може викликати помилки під час виконання. Наприклад, такий код компілюється без помилок:

```
// class type-casting
#include <iostream>

class Dummy {
    float i,j;
public:
    Dummy(){
        i=10.f;j=10.f;
    }
};
```

```
class Addition {
    int x,y;
  public:
    Addition (int a, int b) { x=a; y=b; }
    int result() { return x+y;}
};

int main () {
  Dummy d;
  Addition * padd;
  padd = (Addition*) &d;
  std::cout << padd->result();
}
```

Результат (на різних платформах може бути різним!):

```
0 0
```

Програма оголошує вказівник на Addition, але потім присвоює йому посилання на об'єкт іншого неспорідненого типу за допомогою явного кастингу типу:

```
padd = (Addition*) &d;
```

Необмежене явне приведення типів дозволяє перетворити будь-який вказівник в будь-який інший тип вказівника, незалежно від типів, на які вони вказують. Але в даній програмі подальший виклик результату учасника призведе до помилки під час виконання або інших несподіваних результатів.

Саме тому, на C++ існує декілька варіантів перетворень типів. Для того, щоб контролювати такі типи перетворень між класами, в C++ є чотири конкретні оператори перетворення типу: dynamic_cast, reinterpret_cast, static_cast і const_cast. Їх формат повинен відповідати новому типу, укладеному між кутовими дужками (<>) та одразу після цього, виразом для перетворення між дужками.

- dynamic_cast <new_type> (expression)
- reinterpret_cast <new_type> (expression)
- static_cast <new_type> (expression)
- const_cast <new_type> (expression)

Традиційними перетвореннями типів (type-casting) будуть:

- (new_type) expression
- new_type (expression)

Перетворення типів static_cast

Найбільш популярним перетворенням типів, яке радять використовувати майже всюди, зокрема, в усіх контекстах традиційних перетворень типів, це перетворення static_cast. Його вигляд наступний:

static_cast<Тип> (значення або змінна);

Приклад.

```
int a1 =static_cast<int>(4.578f); // замість (int)4.578f
float b1= static_cast<float>(a1*3);  // замість float(a1*3)
std::cout<<a1<<","<<b1<<std::endl; // 4,12
```

Також static_cast може виконувати всі перетворення, дозволені неявно (не тільки ті, що мають вказівники на класи), а також здатні виконувати протилежні від них. Зокрема:

• Перетворити з void * на будь-який тип вказівника. У цьому випадку гарантується, що якщо значення void * було отримано шляхом перетворення з того самого типу вказівника, результуюче значення вказівника буде однаковим.

• Перетворювати цілі, значення з плаваючою комою та перерахування на типи перерахування.

Приклади:

```
void * ptr = new long long (12LL); //перетворення до void*
 int *ptr2 = static_cast<int*> (ptr); //перетворення з void*
 std::cout<<*ptr2<<std::endl; // 12

 long long * ptr3 = new long long (44LL);
 //int *ptr4 = static_cast<int*> (ptr3); // error: invalid
static_cast from type ‘long long int*’ to type ‘int*’ - неможливе
претворення
 //cout<<*ptr4<<endl;

 int * ptr5 = new int (33);
 //long long *ptr6 = static_cast<long long*> (ptr5); //
неможиве перетворення
 //cout<<*ptr6<<endl;

 void *ptr6 = static_cast<void*> (ptr5);
 // коректне перетворення з int* до long long*

 long long *ptr7 = static_cast<long long*> (ptr6);
 std::cout<<*ptr7<<std::endl;
```

Крім того, static_cast також може виконувати наступне:

• Явно викликати конструктор з одним аргументом або оператор перетворення.

• Перетворити на посилання rvalue (тип, що може бути правою частиною виразу присвоєння).

• Перетворення значень класу перерахування в цілі або значення з плаваючою крапкою.

• Перетворити будь-який тип на void, оцінюючи та відкидаючи значення.

Жодні перевірки не виконуються під час виконання, щоб гарантувати, що об’єкт, який перетворюється, насправді є повноцінним об'єктом типу призначення. Тому програміст повинен забезпечити безпеку перетворення. З іншого боку, він не несе накладних витрат на перевірку безпеки типу динамічного перетворення типів.

```
#include <cstdio>
#include <iostream>

class Base {
    public:
    int x;
    Base(int x_):x(x_){}
};
class Derived: public Base {
    public:
 int y;
   Derived(int x_, int y_):Base(x_),y(y_){}
};

int main(int argc, char **argv){
  Base * a = new Base(1);
  Derived * b = static_cast<Derived*>(a); /* обережно – 
тут може бути помилка!!! */
  std::cout<<"Derived(x,y):"<<b->x<<","<<b->y<<std::endl;

  Base a2(1);
  //Derived b2 = static_cast<Derived>(a2);
 /* неправильне перетворення */
  Derived a3(2,2);
  Base b3 = static_cast<Base>(a3);
  std::cout<<"Base(x):"<< b3.x<<std::endl;
```

```
  Derived * a4 = new Derived(1,1);
  Base * b4 = static_cast<Base*>(a);
  std::cout<<"Base(x):"<< b4->x<<std::endl;
}
```

Результат:

```
Derived(x,y):1,0
Base(x):2
Base(x):1
```

Зауважимо, що

```
Derived * b = static_cast<Derived*>(a);
```

коректний синтаксично код, хоча b вказує на незавершений об'єкт класу та може призвести до помилок виконання під час розіменування вказівника.

Отже, static_cast може виконувати за допомогою вказівників класів не тільки конверсії, дозволені неявно, але й їх протилежні перетворення.

Перетворення типів reinterpret_cast

Перетворення типів reinterpret_cast перетворює будь-який тип вказівника в будь-який інший тип вказівника, навіть неспоріднених класів. Результат операції - це проста двійкова копія значення з одного вказівника на інший. Дозволені всі перетворення вказівників: не вказується ані зміст, ані сам тип вказівника.

Він також може переводити вказівники на цілі типи або з них. Формат, у якому це ціле значення представляє вказівник, залежить від платформи. Єдиною гарантією є те, що вказівник, переданий на цілий тип, достатньо великий, щоб повністю містити його (наприклад, intptr_t), гарантовано зможе повернутися до коректного вказівника.

Конверсії, які можна виконати за допомогою reinterpret_cast, але не static_cast – це операції низького рівня, засновані на реінтерпретації бінарних представлень типів, що в більшості випадків призводить до коду, який є системним і, таким чином, не портативним. Наприклад:

```
class A { /* ... */ };
class B { /* ... */ };
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

Цей код компілюється, хоча це не має особливого сенсу, оскільки тепер b вказує на об'єкт абсолютно неспорідненого і, ймовірно, несумісного класу. Таке перетворення b небезпечно і компілятор в цьому випадку не несе ніякої гарантії, що перетворення відбулося коректно насправді.

Приклад:

```
      int* i;
      char *p = "This is a string";
      i = reinterpret_cast<int*> (p); // перетворення
неспоріднених вказівників
      std::cout << *i;
```

Примітка 1: Багато сучасних керівництв радять не використовувати це перетворення НІКОЛИ!

Примітка 2. Хоча особисто я бачив таке перетворення в деяких відомих бібліотеках.

Перетворення типів const_cast

Цей тип кастингу маніпулює константністю об'єкта, яку вказано вказівником, тобто може встановити його, або навпаки видалити. Наприклад, для передачі константного вказівника на функцію, яка очікує неконстантний аргумент або навпаки:

Приклад:

```
// const_cast
#include <iostream>

void print (char * str){
  std::cout << str << '\n';
}

void func(const int* arr){
  std::cout<<arr[0]<<"ok\n";
}
int main () {
  const char * c = "sample text";
  print( const_cast<char *> (c));

  int* arr = new int[1000];
  func(const_cast<const int*>(arr));
  delete[] arr;
}
```

Результат:

```
sample text
0ok
```

Наведений вище приклад гарантовано спрацьовує, оскільки функція друку не змінює вказаний об'єкт – аргумент функції.

Примітка. Цей тип перетворення теж радять не вживати і єдиний рекомендований випадок — для сумісності зі старими С-подібними інтерфейсами методів та функцій які ви не можете поміняти з якихось причин. Якщо ви однак бачите ситуацію яка “лікується” даним перетворенням, подумайте все ж такі як можна переписати ваш код без цього перетворення.

Для того, щоб описати дію останнього типу перетворень (dynamic_cast) варто розглянуті деякі концепції мови С++.

Оператор typeid

Оператор typeid дозволяє перевіряти тип виразу:

typeid (вираз)

Цей оператор повертає посилання на постійний об'єкт типу type_info, який визначений у стандартному заголовку <typeinfo>. Значення, повернене typeid, можна порівняти з іншим значенням, поверненим typeid за допомогою операторів == та !=, Або може служити для отримання нульової закінченої послідовності символів, що представляє тип даних або ім'я класу, використовуючи його метод name ().

Приклад:

```
// typeid
#include <iostream>
#include <typeinfo>

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))  {
    std::cout << "a and b are of different types:\n";
    std::cout << "a is: " << typeid(a).name() << '\n';
    std::cout << "b is: " << typeid(b).name() << '\n';
  }
}
```

Результат:

```
a and b are of different types:
a is: Pi
b is: i
```

Коли typeid застосовується до класів, typeid використовує RTTI для відстеження типу динамічних об'єктів. Коли typeid застосовується до виразу, тип якого є поліморфним класом, результатом є тип найбільш похідного завершеного об'єкта:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " <<
e.what() << '\n'; }
  return 0;
}
```

Результат:

```
a is: P4Base
b is: P4Base
*a is: 4Base
*b is: 7Derived
```

Примітка 1: рядок, що повертається ім'ям члена type_info, залежить від конкретної реалізації вашого компілятора та бібліотеки. Це не обов'язково простий рядок із типовим ім'ям типу, як у коді бібліотеки компілятору, який використовується для отримання виводу цього прикладу.

Примітка 2: Зверніть увагу, як тип, який typeid вважає для вказівників, є самим типом вказівника (і a, і b є класом типу Base *). Однак, коли typeid застосовується до об'єктів (типу *a та * b), typeid дає їх динамічний тип (тобто тип їх найбільш похідного повного об'єкта).

Якщо оператор typeid виявляє, що він застосовується до вказівника, якому передує оператор взяття посилання (dereference (*)), і цей вказівник має нульове значення, typeid повертає виключення bad_typeid.

Ще один з методів typeinfo — це метод before, який вказує, чи є даний клас наслідником даного типу.

Приклад.

```
// пример использования typeid
#include <iostream>
#include <typeinfo>

class BaseClass {
  int a, b;
   virtual void f() {}; // BaseClass полиморфный
};
class Derived1: public BaseClass {
  int i, j;
};
class Derived2: public BaseClass {
  int k;
};
int main() {
  int i;
  BaseClass *p, baseob;
  Derived1 ob1;
  Derived2 ob2;

  std::cout << "Typeid of i is ";
  std::cout << typeid(i).name() << "\n";
  //typeid  поліморфними типами
  p = &baseob;
  std::cout << "p is pointing to an object of type "; //
Base
  std::cout << typeid(*p).name()<<" and
"<<std::boolalpha<< typeid(*p).before(typeid(baseob)) << "\n";
  p = &ob1;
  std::cout << "p is pointing to an object of type
"<<typeid(*p).name(); // Dderived 1
  std::cout << std::boolalpha << (typeid(*p)==
typeid(Derived1)) <<" and "<<
typeid(*p).before(typeid(baseob))<< "\n";
  p = &ob2;
```

```
  std::cout << "p is pointing to an object of type ";
  std::cout << typeid(p).name() << "\n";
}
```
Результат:
```
Typeid of i is i
p is pointing to an object of type 9BaseClass and false
p is pointing to an object of type 8Derived1true and true
p is pointing to an object of type P9BaseClass
```

Властивість підпорядкування

Підпорядкування - це властивість, що всі об'єкти, які знаходяться в ієрархії класів, повинні виконувати ті самі методи, при цьому об'єкт базового класу може бути замінений об'єктом, що походить від нього (прямо чи опосередковано).
Приклад. Всі ссавці є тваринами (вони походять від них), і всі кішки є ссавцями. Тому, через властивість підпорядкування ми можемо «лікувати» будь-якого ссавця як тварину, а будь-яку кішку як ссавця. Це означає абстракцію, тому що, коли ми "лікуємо" ссавця як тварину, єдина інформація, яку ми повинні знати про це, що вона живе, вона зростає, і т.д., але нічого не стосується саме ссавців.
Це властивість застосовується в C++, коли ми використовуємо вказівники або посилання на об'єкти, які знаходяться в ієрархії класів. Іншими словами, вказівник класу тварин може вказувати на об'єкт класу тварин, ссавців або кішок.

```
#include <iostream>

class Animal {
        public:
        std::string name;

};

class Pet : public Animal{
        public:
         float avgPrice;
};

class Cat : public Pet{
        public:
         int color;
};

int main() {
      Animal* pAnimal1 = new Animal;
```

```
        Animal* pAnimal2 = new Pet;
        Animal* pAnimal3 = new Cat;
        Pet* pPet  = new Cat;

        pAnimal2->name = "Dog";     // коректно
        pPet->avgPrice = 200.0f;   // коректно

        //pAnimal2->avgPrice = 6;  // некоректно
        //pAnimal3->color = 1;   // некоректно

        Cat* pCat = (Cat*)pAnimal3;      // Downcast,
коректно (але не дуже гарно)
        //Cat* pCat = (Cat*)pAnimal2; // тут можуть бути
проблеми
        pCat->color = 1;   // Коректно
    }
```

В останніх рядках прикладу є перетворення вказівника на Animal до вказівника на Cat. Це називається перетворення вниз("Downcast"). Перетворення вниз є корисними і повинні бути використані, але спочатку ми повинні переконатися, що об'єкт, який ми перетворюємо, дійсно є типом, яким ми його передаємо. Перетворення базового класу до незв'язаного класу є помилкою. Щоб вирішити цю проблему, слід використовувати оператори перетворення dynamic_cast <> або static_cast <>. Вони правильно переводять об'єкт з одного класу в інший і викидають виняток, якщо типи класів не пов'язані між собою. напр. Якщо ви спробуєте:

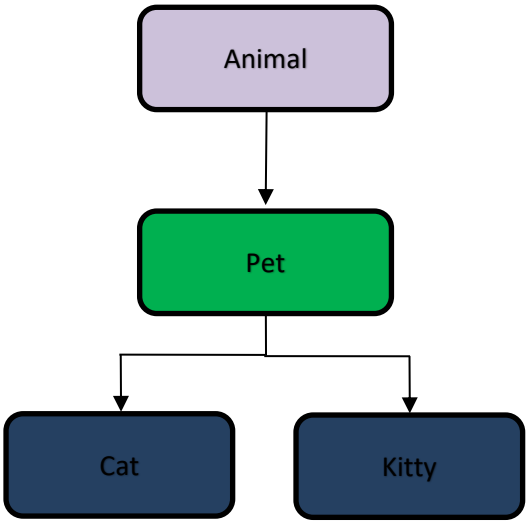
```
    Cat* pC = new Cat;
    Dog* pM = dynamic_cast<Dog*>(pC);
```

Потім програма кине виняток, оскільки Cat не є Dog. Перетворення static_cast не буде виконувати перевірку типу під час компіляції. Тому, якщо у вас є об'єкт, щодо якого ви не впевнені в типі, ви повинні використовувати dynamic_cast і бути готовими до обробки помилок під час перетворення. Якщо ви прибираєте об'єкти, де знаєте типи, ви повинні використовувати static_cast. Головне, не використовуйте старі стилі C перетворень, оскільки вони просто дадуть вам помилку доступу, якщо дані типи не пов'язані між собою.

Перетворення типів вниз та вгору

Перетворення вниз та вгору є важливою частиною C++. Ці перетворення дають можливість будувати складні програми з простим синтаксисом. Цього можна досягти, використовуючи **поліморфізм**.

Мова C++ дозволяє похідному вказівнику класу (або посиланню) трактуватись як вказівник базового класу. Це перетворення вгору. Перетворення вниз – це протилежний процес, який полягає в перетворенні вказівника (або посилання) базового класу на вказівник класу. Перетворення та посилання вниз не слід розуміти як просту трансляцію різних типів даних. Це може призвести до великої плутанини.
У цій темі ми використаємо таку ієрархію класів:



Згідно цієї ієрархії обидва класи Cat та Kitty налажать до Pet та ще більше до Animal. Таким чином, вони обидав мають спільні властивості — вага, ім'я та вік. В той самий час вони мають і відмінності — голос та наявність матері.

Приклад:

```
#include <iostream>

class Animal{
        //content of Animal
       virtual double getAge() {
           return  0;
       }
};


class Pet:public Animal{
public:
        Pet(const char* p_name, int p_age, double
p_weight):
                  name(p_name), age(p_age),weight(p_weight)
{
```

```
            }

            std::string name;
            int age;
            double weight;

            void show() {
                    std::cout << "Name: " << name << " age: "
<< age << " weight: " << weight<< "\n";
            }

            void addYear(double added_weight){
                    age++;
                    weight += added_weight;
            }

    };

    class Cat : public Pet{
    public:
            Cat(const char* p_name, int c_age, double
c_weight):
                                                    Pet(p_name,
c_age, c_weight){
                voice = "myaw";
            }
            std::string voice;
            std::string getMurr()        {
                    return voice;
            }
            double getAge() {
               return  age;
          }
    };

    class Kitty : public Pet{
    public:
            Kitty(const char* c_name, int c_age, double
c_weight, Cat* k_mother) : Pet(c_name, c_age, c_weight){
         mother = k_mother;
            }
            Cat* mother;
```

```
            Cat* getMother()        {
                    return mother;
            }
            double getAge() {
               return  0.5;
          }
    };

    void new_year(Pet* pet){
            std::cout << "Happy New Year!!!\n";
            pet->addYear(2.0);
            pet->show();
    };

    int main(){
        //Вказівник на базовий обєкт: pointer to base class
object
        Pet* pussy;

        // Обєкти похідних класів: object of derived class
        Cat m1("Murka", 3, 4.5);
        Kitty c1("Malka",0, 1.5, &m1);

        // перетворення вгору: implicit upcasting
        pussy = &m1;

        //It's ok
        std::cout<<pussy->name<<"\n";
        std::cout<<pussy->age<<"\n";

        //Помилка(переторення вгору) Fails because upcasting
is used
        //std::cout<<pussy->getAge();

        new_year(&c1);
        new_year(&m1);

        std::cout<<"Mother of "<<c1.name<<" is
"<<c1.getMother()->name;

      //explicit downcasting from Pet to Cat
       Cat* m2 = (Cat*)(pussy);
```

```
    Pet p1("Tom", 7, 5.6);
    //try to cast an Pet to Cat
   Cat* m3 = (Cat*)(&p1);
   std::cout << m3->getAge() << "\n";

    Pet p2("Vas'ka", 4, 4.4);
    Cat* m4 = dynamic_cast<Cat*>(&p2);
    if (m4)
         std::cout << m4->getAge() << "\n";
    else
         std::cout << "Can't  cast from Pet to Cat\n";
}
```

Cat та Kitty мають спільну частину яка залишається незмнним при перетворенні. Слід відзначити, що і перетворення, і присвоєння не змінюють об'єкт сам по собі. Під час використання ви просто "позначаєте" об'єкт різними способами.

Перетворення вгору: Upcasting

Перетворення вгору(Upcasting) **- це процес обробки вказівника або посилання на похідний об'єкт класу як вказівник базового класу**. Не потрібно здійснювати перетворення вручну. Просто потрібно призначити похідний вказівник класу (або посилання) на вказівник базового класу:

```
        //Вказівник на базовий обєкт: pointer to base class
object
        Pet* pussy;

        // Обєкти похідних класів: object of derived class
        Cat m1("Murka", 3, 4.5);
        Kitty c1("Malka",0, 1.5, &m1);

        // перетворення вгору: implicit upcasting
        pussy = &m1;
```

Коли ви використовуєте це перетворення, об'єкт не змінюється. Тим не менш, коли ви оновлюєте об'єкт, ви матимете доступ до лише функцій-членів (методів) та членів даних, визначених у базовому класі:

```
    //It's ok
        std::cout<<pussy->name<<"\n";
        std::cout<<pussy->age<<"\n";
    / //Помилка(переторення вгору) Fails because upcasting is
used
        //std::cout<<pussy->getAge();
```

Приклад використання перетворення

Однією з найбільших переваг перетворення є можливість запису загальних функцій для всіх класів, що походять з одного базового класу. Подивіться на приклад:

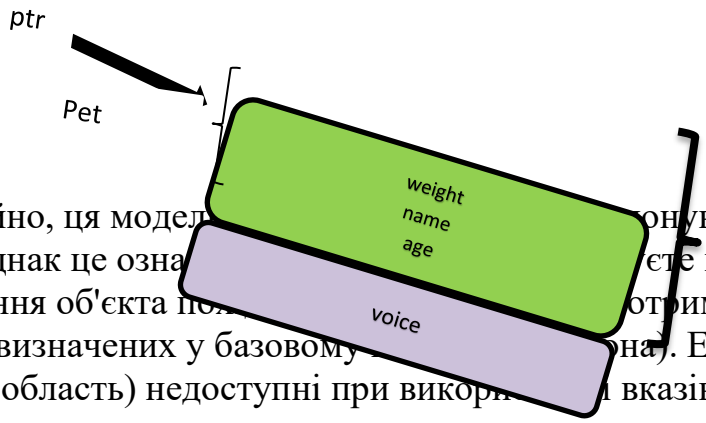
```
void new_year(Pet* pet){
        std::cout << "Happy Birthday!!!\n";
        pet->addYear(2.0);
        pet->show();
};
```

Ця функція буде працювати з усіма класами, що походять від класу Співробітник. Коли ви називаєте його об’єктами типу Cat та Kitty, вони будуть автоматично перенесені на клас Pet:

```
//Автоматичне перетворення вгору: automatic upcasting
  new_year(&c1);
  new_year(&m1);
```

Модель пам'яті

Як відомо, похідний клас розширює властивості базового класу. Це означає, що похідний клас має властивості (члени даних та функції членів) базового класу та визначає нових членів даних та функцій членів. Подивіться на макет пам'яті класів Pet та Cat:



Звичайно, ця модел онува пам'яті для об'єктів. Однак це озна є в Cat азового класу для наведення об'єкта по отримати доступ лише до елементів, визначених у базовому на). Елементи похідного класу (сіра область) недоступні при викори вказівника базового класу.

Перетворення вниз: Downcasting

Перетворення вниз (Downcasting) – протилежний до перетворення вгору процес перетворення. Воно перетворює вказівник базового класу у похідний вказівник

класу. Таке перетворення потрібно робити вручну, тому що вам при цьому перетворенні потрібно вказати чіткий тип перетворення. Перетворення вниз не є безпечним, як перетворення вгору. Ви знаєте, що похідний об'єкт класу завжди може розглядатися як об'єкт базового класу. Однак обернене твердження не є вірним. Наприклад, в нашому прикладі Cat – це завжди Pet, але Pet не завжди є Cat. Це може бути й Kitty.
Ви повинні використовувати явне перетворення для перетворення вниз:

```
//explicit downcasting from Pet to Cat
Cat* m2 = (Cat*)(pussy);

Pet p1("Tom", 7, 5.6);
//try to cast an Pet to Cat
Cat* m3 = (Cat*)(&p1);
std::cout << m3->getAge() << "\n";
```

Цей код збирається та працює без будь-яких проблем, оскільки emp вказує на об'єкт класу Manager.

Що станеться, якщо ми спробуємо схилити вказівник базового класу, який вказує на об’єкт базового класу, а не на об'єкт похідного класу? Спробуйте скласти і запустити цей код:

```
Pet p1("Tom", 7, 5.6);
//try to cast an Pet to Cat
Cat* m3 = (Cat*)(&p1);
std::cout << m3->getAge() << "\n";
```

Об'єкт e1 не є об'єктом класу Cat. Він не містить жодної інформації про voice. Ось чому така операція може дати неочікувані результати.

Якщо ви спробуєте скинути вказівник базового класу (Pet), який насправді не вказує на об'єкт похідного класу (Cat), ви отримаєте доступ до пам'яті, яка не має жодної інформації про похідний об'єкт класу (жовта область). Це головна небезпека помилок.

Ви можете використовувати безпечне перетворення, яке допоможе вам дізнатися, чи можна один тип правильно перетворити на інший. Для цього потрібне динамічне перетворення типів (dynamic_cast).

Динамічне перетворення типів: dynamic_cast

Динамічне перетворення типів (dynamic_cast) – це оператор, який безпечно перетворює один тип в інший. У випадку, якщо це перетворення можливе та безпечне, воно повертає адресу об'єкта, який перетворюється. В іншому випадку воно повертає nullptr.

Синтаксис:

dynamic_cast<тип> (об’єкт)

Якщо ви хочете використовувати dynamic_cast для перетворення вниз, базовий клас повинен бути поліморфним - він повинен мати принаймні одну віртуальну функцію. Тобто, потрбіно додати віртуальну функцію:

```
virtual void foo() {}
```

Тепер ви можете використовувати перетворення вниз для перетворення вказівників класу Cat у вказівники похідних класів.

```
Pet p2("Vas'ka", 4, 4.4);
Cat* m4 = dynamic_cast<Cat*>(&p2);
if (m4)
    std::cout << m4->getAge() << "\n";
else
    std::cout << "Can't  cast from Pet to Cat\n"
```

У цьому випадку динамічне перетворення типів повертає нульовий вказівник (nullptr). Тому ви побачите попереджувальне повідомлення:

```
warning: dynamic_cast of ‘Pet p2’ to ‘class Cat*’ can never succeed

  Cat* m4 = dynamic_cast<Cat*>(&p2);
```

Розглянемо тепер ситуацію, коли динамічне перетворення типу має сенс. Нехай в нас є компанія, яка складається зі співробітників різних типів: (віртуальний клас Employee, його наслідники Programmer, QA, Menager). Нехай нам потрібно завести масив співробітників, а потім вивести всіх програмістів.

```
#include <iostream>
const size_t NUM_EMPLOYEES = 4;
class Employee {
  public:
    Employee () { std::cout << "Employee\n"; }
    virtual void print() = 0;
};
class Programmer : public Employee {
  public:
    Programmer() { std::cout << "Programmer\n"; }
    void print () { std::cout << "Printing programmer\n";
}
};
class QA : public Employee {
  public:
```

```
   QA () { std::cout << "QA\n"; }
   void print() { std::cout << "Printing QA\n"; }
};

class Menager : public Employee {
  public:
   Menager () { std::cout << "Menager\n"; }
   void print () { std::cout << "Printing menager\n"; }
};

int main() {
  Programmer prog1, prog2;
  Menager ex;
  QA sp;  //
  Employee *e[NUM_EMPLOYEES];
 // ініціалізуємо масив
  e[0] = &prog1;
  e[1] = &sp;
  e[2] = &ex;
  e[3] = &prog2;
  // цикл по робітникам:
  for (int i = 0; i < NUM_EMPLOYEES; i++) {
   Programmer *pp = dynamic_cast<Programmer*>(e[i]);
   if(pp) {
     std::cout << "Is a programmer\n";
     pp->print();
   }
   else {
    std::cout << "Not a programmer\n";
   }
 }
}
```

Результат роботи:

```
Employee
Programmer
Employee
Programmer
Employee
Menager
Employee
QA
Is a programmer
Printing programmer
```

```
Not a programmer
Not a programmer
Is a programmer
Printing programmer
```

7. Простори імен (Namespaces)

Потреба в просторах імен та їх визначення

Реальні програми або програми складаються з багатьох вихідних файлів. Ці файли можуть бути авторськими та підтримуватися більш ніж одним розробником або програмістом. Врешті-решт, окремі файли організовуються та зв'язуються для отримання остаточної програми.
Традиційно організація файлів вимагає, щоб усі імена, які не інкапсульовані у визначеному просторі імен (наприклад, у тілі функції чи класу чи блоку перекладу), повинні міститись в одному спільному глобальному просторі імен, тобто бути доступними для всіх прогам та модулів даних файлів ід тими самими іменами. Отже, під час пов'язування окремих модулів може виникнути кілька однакових визначень імен або зіткнень імен.
Механізм простору імен у C++ долає проблему співпадінь імен у глобальному масштабі. Механізм простору імен дозволяє програмі розподілити на кількість підсистем. Кожна підсистема може визначати та функціонувати в межах своєї сфери.
Декларація просторів імен ідентифікує та присвоює унікальне ім'я оголошеному користувачем простору імен. Це буде використано для вирішення зіткнення чи конфлікту імен при великій розробці програм і бібліотек, де є багато програмістів або розробників, які працюють у різних частинах програми.
Для використання просторів імен C++ передбачено два етапи:
1. Однозначно ідентифікувати простір імен за допомогою ключового слова namespace.
2. Отримати доступ до елементів ідентифікованого простору імен, застосувавши за допомогою ключового слова using або назви простору імен та оператору доступу(::).

Загальна форма визначення простору імен:
namespace <indentifier>{
//тіло простору імен
}

Приклад:
```
namespace  NewOne{
      int p;
```

```
        long q;
        class A { …};
        void foo {…};
}
```
де p та q звичайні змінні, A – визначений користувачем клас, а foo – деяка функція, але всі вони інтегровані в межах простору імен NewOne. Для доступу до цих змінних за межами простору, потрібно використовувати оператор доступу (scope operator) ::. Зокрема, для попереднього прикладу:
```
NewOne::p;
NewOne::q;
NewOne::A;
NewOne::foo;
```
Приклад:
```
#include <iostream>

namespace  NewOne{
        int p;
        long q;
        struct A {
           static void print(){
           std::cout<<"A";
           }
        };
        void foo(int x) {
          std::cout<<"x="<<x;
        };
}

int main(){
  std::cout<< NewOne::p;
  std::cout<<NewOne::q;
  NewOne::A x;
  x.print();
  NewOne::A::print();
  NewOne::foo(42);
}
```

Визначення простору імен може бути вкладено в інше визначення простору імен. Кожне визначення простору імен повинно з'являтися або в межах файлу, або відразу в іншому визначенні простору імен. Використати простори імен можна як за допомогою ключового слова using так і за допомогою оператору доступу ::.

Приклад:

```
//  Створення просторів імен
#include <iostream>
using namespace std; // підключаємо стандартний простір імен

namespace SampleOne{ // перший простір імен
	float p = 10.34;
}

namespace SampleTwo{ // другий простір імен

	using namespace SampleOne; // використовуємо перший простір всередині другого
	float q = 77.12;

	namespace InSampleTwo   { // вкладений простір імен
		float r = 34.725;
	}
}

int main(){
	// ця директива підключає декларації змінних з SampleTwo
	using namespace SampleTwo;

	// ця директива підключає лише декларації змінних внутрішнього простору  InSampleTwo
	using namespace SampleTwo::InSampleTwo;

	// локальна декларація, має пріоритет
	float p = 23.11;
	cout<<"p = "<<p<<endl;
	cout<<"q = "<<q<<endl;
	cout<<"r = "<<r<<endl;
}
```

Результат роботи:

```
C++ Namespace using directive
```

Приклад 2 (без директиви using) :

```
// використання простору імен без директиви using
#include <iostream>
using namespace std;
```

```
namespace NewNsOne{
	// декларація змінної в NewNsOne
	int p = 4;
	// декларація функції в NewNsOne
	int funct(int q);
}

namespace NewNsTwo{
	// декларація змінної в NewNsTwo
	int r = 6;

	// декларація функції в NewNsTwo
	int funct1(int numb);

	// декларація вкладеного простору імен
	namespace InNewNsTwo	{
		// декларація змінної в InNewNsTwo
		long tst = 20.9456;
	}
}

int main(){
	/* наступний код згенерує помилку  тому що це не загальний
простір імен а  головна функція  main */
     // namespace local {
	//	int k;
	// }

	cout<<"NewNsOne::p is "<<(NewNsOne::p)<<endl;
	cout<<"NewNsTwo::r is "<<(NewNsTwo::r)<<endl;
	cout<<"NewNsTwo::InNewNsTwo::tst		is"		<<
(NewNsTwo::InNewNsTwo::tst)<<endl;
}
```

Результат роботи:

```
C++ Namespace without using directive
```

Примітка 1. Всі стандартні функції, константи та глобальні змінні, що використовуються в C++ починаючи з 98-го стандарту визначені та містяться в стандартному глобальному просторі імен. Тому для використання, наприклад,

методу потокового виведення cout<< нам потрібно або підключити для нашої програми за допомогою директиви using стандартний простір імен std:

```
using namespace std;
… //
cout<<  “Using cout”;
```

або кожен раз при використанні класу cout вказувати за допомогою оператору доступу(::) що він належить стандартному простору імен std:

```
std::cout<<“Using cout”;
```

Примітка 2. Потреба у якихось інших просторах імен, що відмінні від стандартних може зустрітись коли ми використовуємо якусь велику нестандартну бібліотеку, наприклад, boost, llvm, opencv і таке інше.

Псевдоніми просторів імен (Namespace Alias)

Альтернативне ім’я може використовуватися для позначення ідентифікатору простору імен. Псевдонім корисний, коли вам потрібно спростити довгий ідентифікатор простору імен.

Приклад:

```
// псевдоніми просторів імен namespace alias
#include <iostream>

namespace TheFirstLongNamespaceSample{
  float p = 23.44;
   namespace TheNestedFirstLongNamespaceSample {int q = 100; }
}
// псевдонім: an alias namespace
namespace First = TheFirstLongNamespaceSample;
// псевдонім для вкладеного простору імен
namespace                          Second                          =
TheFirstLongNamespaceSample::TheNestedFirstLongNamespaceSample
;

int main(){
  using namespace First; // використання першого псевдоніму
  using namespace Second; // використання другого псевдоніму
  std::cout<<"p = "<<p<<endl;
  std::cout<<"q = "<<q<<endl;
```

```
}
```

Результат роботи:
```
C++ Namespace alias
```

Розширення просторів імен (Namespace Extension)

Визначення простору імен можна розділити на кілька частин однієї одиниці трансляції.
Якщо ви оголосили простір імен, ви можете розширити вихідний простір імен, додавши нові декларації. Будь-які розширення, які вносяться до простору імен після використання заяви, не будуть відомі в точці, в якій відбувається декларація використання.

```
// оригінальний простір імен
namespace One{
       // змінні цього простору імен
   //*** означення в цьому просторі
}

 namespace Two{
       //** означення в новому просторі
}

// Розширення простору One
namespace One{
       // декларація функції простору One  - додані функції та
класи..
      int x;
      void funct1();
       int funct2(int p);
      class A{***}
}
```

Приклад:
```
//розширення просторів
#include <iostream>

struct SampleOne
{ };
struct SampleTwo
{ };
```

```
// оригінальний простір імен namespace
namespace NsOne{
  // оригінальна функція
  void FunctOne(struct SampleOne){
    std::cout<<"Processing struct..."<<endl;
  }
}
using NsOne::FunctOne; // використання using-declaration...
// розширення простору NsOne
namespace NsOne{// перевантажена (overloaded) функція...
void FunctOne(SampleTwo& x){
  std::cout<<"Processing function..."<<endl; }
}
int main(){
  SampleOne TestStruct;
  SampleTwo TestClass;
  FunctOne(TestStruct);
  // Ця функція не запрацює, бо нема її перевантаженого екземпляру
  // FunctOne(TestClass);
}
```

Результат роботи:

```
C++ Namespace extension
```

Анонімні простори імен (Unnamed/anonymous Namespace)

Можна використовувати простір імен ключових слів без ідентифікатора перед заключною дужкою. Це може бути кращою альтернативою використанню глобальної статичної декларації змінної.
Кожен ідентифікатор, який укладений у неназваному просторі імен, є унікальним у блоці трансляції, у якому визначено неназваний простір імен.

Синтаксис:
namespace { namespace_body }

Цей код поводиться так само як:

```
Namespace anonymous { namespace_body}
using namespace anonymous;
```

Приклад:

```
// анонімний простір імен
#include <iostream>

// Простір імен без назви (anonymous namespace)
namespace{
   int p = 1; // anonymous::p
}

void funct1(){
  ++p; // anonymous::++p
}

namespace One{
  // вкладений анонімний простір
  namespace{
    int p;
    // One::anonymous::p
    int q = 3; // One::anonymous::q
   }
}
// використання using-declaration
using namespace One;
void testing(){
  // ++p;
  // помилка: anonymous::p або One::anonymous::p?
  // One::++p; // помилка , One::p - невизначене (undefined)
  std::cout<<"++q = "<<++q<<"\n";
}

int main(){
  testing();
}
```

Результат роботи:

```
++q = 4
```

Доступ до елементів простору імен (Accessing Namespace Elements)

Існують такі шляхи доступ до елементів простору імен:

- Використовуючи повний шлях імені
- Використання директиви using

- Використання декларації using

Використання повного шляху імені

В цьому варіанті (найбільш уживаному та тому що радять до використання більшість практичних керівництв) потрібно просто вказувати повний шлях імені розділяючи імена за допомогою оператору доступу(::):
<Простір>:: [<Підпростір>::] <ідентифікатор>

Приклад:

```
namespace NewNsOne{
      int p = 4;
      int funct(int q){
         std::cout<<"q="<<q;
      }
}

namespace NewNsTwo{
      int r = 6;
      int funct1(int numb);
      namespace InNewNsTwo      {
             long tst = 20.9456;
      }
}

 int main(){
     std::cout<<"NewNsOne::p is
"<<(NewNsOne::p)<<(NewNsOne::funct(2))<<"\n";
     std::cout<<"NewNsTwo::r is "<<(NewNsTwo::r)<<"\n";
std::cout<<"NewNsTwo::InNewNsTwo::tst is"<<
(NewNsTwo::InNewNsTwo::tst)<<"\n";
}
```

Директива using

Цей метод корисний, коли ви хочете отримати доступ до кількох або всіх членів простору імен. Директива using вказує, що всі ідентифікатори в просторі імен знаходяться в області застосування в тому місці, коли зроблено оператор-використання.
Цей метод також є транзитивним; це означає, що коли ви застосовуєте директиву using до простору імен, яка містить використання директиви всередині себе, ви також отримуєте доступ до цих просторів імен.
Синтаксис:

using namespace <ім'я простору>;

Приклад:

```
// Директива using
#include <iostream>
using namespace std;
namespace One
{    float p = 3.1234;    }

namespace Two {
       using namespace One;
      float q = 4.5678;
      namespace InTwo
       {    float r = 5.1234;    }
}

int main(){
       // використання всіх змінних з Two
       using namespace Two;

       // доступ лише до вкладеного підпростору імен InTwo
       using namespace Two::InTwo;

      // локальна змінна
       float p = 6.12345;

       cout<<"p = "<<p<<endl;
       cout<<"q = "<<q<<endl;
       cout<<"r = "<<r<<endl;
}
```

Результат:

```
 p = 6.12345
q = 4.5678
r = 5.1234
```

Декларація using

Ви можете отримати доступ до елементів простору імен індивідуально, застосувавши using-декларацію. Тут ви додаєте оголошений ідентифікатор до

локального простору імен. Така using-декларація дозволяє використовувати лише окремі функції, змінні або класи з простору імен.

Синтаксис:
using::<ідентифікатор змінної, класу або функції>;

Приклад:

```
// Декларція using :
// функція funct2() визначена в двох різних просторах

#include <iostream>

using std::cout;

namespace NSOne{

       float funct1(float q) {
             return q;
         }
      // перше визначення funct2()
      void funct2()  {
             cout<<"namespace NSOne, funct2() function\n";
      }
 }

namespace NSTwo{
       // друге визначення funct2()
       void funct2(){
        cout<<"namespace NSTwo, funct2() function\n";    }
       }

int main(){
      // використання using визначає версію funct2()
      using NSOne::funct1;     // визначає версію
      using NSTwo::funct2;     // визначає версію

      float p = 4.556;           // локальна змінна
      cout<<"First p value, local = "<<p<<"\n";
      p = funct1(3.422);

      cout<<"Second p value, by function call = "<<p<<"\n";
      funct2();
```

```
}
```
Результат:
```
First p value, local = 4.556
Second p value, by function call = 3.422
namespace NSTwo, funct2() function
```

8. Шаблони функцій та шаблони класів

Потреба в шаблонах функцій та шаблонах класів

Одним з недоліків мов програмування з типізацією є те, що коли визначається певна функція потрібно жорстко фіксувати для яких типів ця функція визначена та який результат вона повертає. Насправді, звичайно, це інколи гарно — бо гарантує виконання функції лише власне для тих типів для яких вона призначена. Але інколи, як наприклад для функції

```
int max (int a, int b){
     return a>b?a:b;
}
```

було б непогано зробити так, щоб не було потреби визначати її окремо і для інших стандартних типів — long long, unsigned, string і так далі, а також можливо й для деяких нестандартних (для тих з них, де визначена операція порівняння), бо ми бачимо, що код функції max для них буде такий самий.

Для тих з типізованих мов, що є об'єктно-орієнтовними та всі типи є нащадками якогось базового класу можна спробувати реалізувати цю функцію для всіх типів з допомогою перетворення вниз, але в C++ для стандартних типів така можливість відсутня.

Ще одна причина появи синтаксису шаблонів є створення класів для реалізації стандартних контейнерів даних, наприклад, стеку.

При створенні таких структур даних на C завжди потрібно вказувати який тип даних в цій структурі використовується, наприклад:

```
struct Stack_{
   int data;
  struct Stack_ * next;
} Stack;
```

А отже, якщо потрібно визначити таку саму структуру з тими ж самими функціями (методами) потрібно знов створювати нову структуру та переписувати той самий код з іншим типом, що, звичайно, є погано.

Звичайно, навіть синтаксис мови C дозволяє обійтись без повторення коду за допомогою певних хитрощів, але такі варіанти є достатньо нетривіальними, тому починаючи з 98-го стандарту додали нову властивість синтаксису, яка зветься шаблони.

Шаблони функцій

Шаблон функції – це опис функції, яка залежить від даних довільного типу. Під час виклику такої функції компілятор автоматично проаналізує тип фактичних аргументів, згенерує для них програмний код. Це називається неявним створенням **екземпляру шаблону**.

Запис:

template <class Тип1, class Тип2,…, class ТипN>

Тип Функції НазваФункції(Тип1 аргумент1, Тип2 аргумент2,…,ТипN аргументN);

або

template <typename Тип1, typename Тип2,…, typename ТипN>

Тип_Функції НазваФункції(Тип1 аргумент1, Тип2 аргумент2,…,ТипN аргументN);

це оголошення шаблону.

В цьому визначенні:

- template - ключове слово, що вказує на визначення шаблону,
- Тип1, …, ТипN – назви деяких узагальнюючих типів.
- class або typename - ключові слова, що вказують на те що цей тип є узагальненим.
- Тип_Функції НазваФункції(Тип1 аргумент1, Тип2 аргумент2,…,ТипN аргументN) – опис визначення функції в рамках синтаксису Сі++. У списку формальних параметрів можуть бути присутні як параметри узагальнюючих типів так і стандартні або користувацькі типи.

Якщо шаблон описаний перед головною програмою, то він, як і звичайна функція, оголошення не потребує.

Підхід у програмуванні, що грунтується на використанні шаблонів функцій називається **узагальнюючим програмуванням**.

Примітка. Ключові слова class та typename абсолютно рівносильні (синоніми) та немає ніякої різниці яким з них користуватись.

Примітка до примітки. Насправді колись була домовленість використовувати class коли передбачалося використовувати саме класи, а typename — коли потрібно підставляти лише стандартні типи, але в стандарті це навіть не залишилося в якості рекомендацій.

Приклад:

```
#include <iostream>
#include <cstring>

template <typename T> T maxArray(const T* array, size_t size){
  T max = array[0];
  for (size_t i = 1; i < size; i++)
     if (max < array[i])
```

```
            max = array[i];
  return max;
}
int main(){
  char array [] = "aodsiafgerkeio";
  int len = strlen(array);
  std::cout << "Максимальний елемент масиву типу char: " <<
maxArray(array, len) << "\n";
  int iArray [5] = {3,5,7,2,9};
  std::cout << " Максимальний елемент масиву типу int: " <<
maxArray(iArray, 5) << "\n";
}
```

Результат:

```
Максимальний елемент масиву типу char: s
Максимальний елемент масиву типу int: 9
```

Тут у нас в шаблоні функції використовувалися вбудовані типи даних, тому в рядку ми написали template <typename T> (або <class T>). Замість T можна підставити будь-яке інше ім'я, яке є коректним ідентифікатором.

У рядку template <typename T> T maxArray(const T* array, int size) виконується визначення шаблону з одним параметром - T, причому цей параметр буде мати один з вбудованих типів даних, так як вказано ключове слово typename.

Нижче, оголошена функція, яка відповідає всім критеріям оголошення звичайної функції, є заголовок, є тіло функції, в заголовку є ім'я і параметри функції, все як завжди. Але що цю функції перетворює в шаблон функції, так це параметр з типом даних T, це єдиний зв'язок з шаблоном, оголошеним раніше. Якби ми написали

```
int maxArray(const int* array, int size){
    int max = array[0];
    for (int i = 1; i < size; i++)
     if (max < array[i])
           max = array[i];
    return max;
}
```

то отримали б звичайну цілу функцію від цілих аргументів,

Аналогічно, при T рівним std::string отимаємо

```
std::string maxArray(const std::string* array, int size){
    std::string max = array[0]; // максимальне значення в масиві
    for (int i = 1; i < size; i++)
     if (max < array[i])
          max = array[i];
```

```
    return max;
}
```

Тобто, T — це навіть не тип даних, а зарезервоване місце під будь-який вбудований тип даних. Тобто коли виконується виклик цієї функції, компілятор аналізує параметр шаблону функції і створює екземпляр для відповідного типу даних: int, char і так далі.

Примітка. Слід розуміти, що навіть якщо обсяг коду менше, то це не означає, що пам'яті програма буде споживати менше. Компілятор сам створює локальні копії функції-шаблону і відповідно пам'яті споживається стільки, як якщо б ви самі написали всі екземпляри функції, як у випадку з перевантаженням.

Відзначимо тієї факт, що поки немає виклика функції-шаблоном, при компіляції вона в бінарному коді не створюється (**не інстанціюється**). А якщо оголосити групу викликів функції зі змінними різних типів, то для кожного компілятор створить свою реалізацію на базі шаблону.

Виклик шаблонної функції, в загальному випадку, відповідає виклику звичайної функції. В цьому випадку компілятор визначить, який тип використовувати замість типу T, на підставі визначення типів фактичних параметрів. Але якщо підставлені параметри виявляться різних типів, то компілятор не зможе вивести (**інстанціювати шаблон**) реалізацію шаблону. Так, в нижче наступному коді компілятор “спіткнеться” на третьому виклику, так як не може визначити, чому дорівнює Type:

```
#include <iostream>
template<class Type>
Type _min(Type a, Type b) {
    return (a < b) ?a:b;
}

int main(int argc, char** argv) {
    std::cout << _min(1, 2) << std::endl;
    std::cout << _min(3.1, 1.2) << std::endl;
    std::cout << _min(5, 2.1) << std::endl; /* error: no matching function for call to ‘_min(int, double)’*/
    }
```

Вирішується ця проблема вказанням конкретного типу при виклику функції.

```
int main(int argc, char** argv) {
    std::cout << _min<double>(5, 2.1) << std::endl;
}
```

Примітка. Не всі шаблони завжди можуть бути коректно інстанційовані. Дійсно, компілятор просто підставляє потрібний тип в шаблон. Але чи завжди

отримувана функція буде працездатна? Очевидно, що ні. Будь алгоритм може бути визначений незалежно від типу даних, але він обов'язково користується властивостями цих даних. У випадку з шаблонної функцією _min це вимога визначення оператора упорядкування (оператор <).

Будь-який шаблон функції передбачає наявність певних властивостей параметризованого типу, в залежності від реалізації (наприклад, оператору копіювання, оператору порівняння, наявності певного методу і т.д.). В очікуваному стандарті мови C++ за це будуть відповідати концепції.

Ще один, більш складний приклад. Визначимо шаблон функції від трьох типів:

```
#include <iostream>
template <class T1, class T2, class T3>
T3 strange_mul(T1 x, T2 y) {
    T3 res = static_cast<T3>(x) * static_cast<T3>(y);
    return res;
}
```

Тут ми визначили шаблон функції, який робить множення двох змінних різних типів, спочатку зводячи їх до третього типу. Ця функція буде коректно інстанційована, наприклад, наступними визначеннями типів та викликами:

```
long long y1 =  strange_mul<int, unsigned,long long>(1,2);
double y2 =  strange_mul<float, int, double>(1.0f,2);
```

Але у наступних випадках, вона повинна «впасти» при виконанні:

```
double y3 =  strange_mul<float, string, double>(1.0f,"2");
//error: invalid static_cast
string y4 =  strange_mul<string, string, string>("1","2"); //
error: no match for ‘operator*’
```

В першому випадку, вона «впаде», бо не зможе виконати перетворення типу static_cast<double>, а в другому бо не зможе виконати множення для типу string.

В усіх схожих випадках відповідальність за коректність шаблону кладеться на плечі програміста, бо програму буде стикатись з помилкою безпосередньо на етапі виконання.

Перевантаження шаблону функції

Шаблони функцій також можна перевантажувати іншими шаблонами функцій, змінивши кількість переданих параметрів в функцію. Ще однією особливістю перевантаження є те, що шаблонні функції можуть бути перевантажені зазвичай не шаблонними функціями. Тобто вказується те саме ім'я функції, з тими ж параметрами, але для певного типу даних, і все буде коректно працювати.

Шаблони функцій також можуть перевантажуватися.

Зазвичай дана перевантаження виконується при довизначенні шаблонів до вказівників, як на прикладі.

```
template<class Type> Type* _min(Type* a, Type* b){
    return (*a < *b)?*a:*b;
        }
```

Іншим варіантом, коли треба перевантажити шаблон — це спеціалізація шаблонної функції.

У деяких випадках шаблон функції є неефективним або неправильним для певного типу. В цьому випадку можна спеціалізувати шаблон, - тобто написати реалізацію для даного типу. Наприклад, у випадку з рядками можна вимагати, щоб функція порівнювала тільки кількість символів. У разі спеціалізації шаблону функції тип, для якого уточнюється шаблон в параметрі не вказується. Нижче наводиться приклад зазначеної спеціалізації.

```
template<>
std::string _min(std::string a, std::string b){
    if(a.size() < b.size()){
        return a;
    }
    return b;
}
```

Спеціалізація шаблону для конкретних типів робиться знову ж з міркування економічності: якщо ця версія шаблону функції в коді не використовується, то вона не буде включена в бінарний код.

Примітка. Для параметрів шаблону, так само як для параметрів функції є можливість задавати параметри за замовченням, щоправда для функцій ця можливість додана лише зі стандарту C++11.

Приклад

```
#include <iostream>
template <typename T=int> T func(T x, int y){
    T res = x / y;
    return res;
}
int main(){
int x=11,y=2,z;
double a=11.0,b,c;
z  = func(x,y);  //  5 - ціле
b = func<double>(x,y); // 5.5 - дійсне
c= func(a,y); // 5.5 - дійсне
}
```

Стандартні шаблони функцій

Деякі популярні шаблони функцій вже присутні в стандартній бібліотеці C++, зокрема:

1) Шаблон максимуму (бібліотека <algorithm.h>):

```
template<class T>
```

```
const T& max(const T& a, const T& b){
    return (a < b) ? b : a;
}
```

Використання:

```
int z = std::max(1,2);
 double x=1.0, y=2.0;
 y = std::max(x,y);
 std::string                          zs                          =
std::max(std::string("A"),std::string("ABC"));
```

2) Шаблон мінімуму (бібліотека <algorithm.h>):

```
int z1 = std::min(1,2);
```

3) Шаблон функції обміну значеннями двох змінних (C++98: <algorithm>, C++11: <utility>)

```
// swap algorithm example (C++98)
#include <iostream>     // std::cout
#include <algorithm>    // std::swap

int main () {
  int x=10, y=20;    // x:10 y:20
  std::swap(x,y);    // x:20 y:10
}
```

Шаблони класів

Аналогічно як при створенні функцій, так і при розробці класів для різних типів даних, потрібно писати програмний код для кожного типу окремо. При цьому часто методи і операції над даними різних типів можуть містити один і той же повторюваний код. Щоб уникнути повторюваності написання коду для різних типів даних, в мові C++ використовуються так звані **шаблони (templates) класів**.

Шаблон класу дозволяє оперувати даними різних типів в загальному випадку. Тобто, немає прив'язки до певного конкретного типу даних (int, float, ...). Вся робота виконується над деякими узагальненим типом даних, наприклад типом з ім'ям T.

Фактично, оголошення шаблону класу є тільки описом. Створення реального класу з заданим типом даних виконується компілятором в момент компіляції, коли об’являється об’єкт класу.

Загальна форма декларації шаблонного класу має наступний вигляд:

ClassName <*types*> *objName***;**

де

- *ClassName* – ім’я шаблонного класу;
- *types* – типи даних в програмі;

•*objName* – ім’я об’єкту (екземпляру) класу.

Ключове слово class може бути замінено на слово typename. Тоді загальна форма декларації шаблонного класу може бути наступною:

```
template <typename T1, typename T2, …, typename Tn >
class ClassName {
   // тіло класу
   // ...
}
```

або

```
template <class T1, class T2, …, class Tn> class ClassName
{
   // тіло класу
   // ...
}
```

де

•T1, T2, …, Tn – узагальнені імена типів, які використовуються в класі;

•ClassName – ім’я класу.

Згідно стандарту немає різниці між ключовими словами class та typename.

Примітка. Колись була домовленість, що ключове слово typename говорить про те, що в шаблоні буде використовуватися вбудований тип даних, такий як: int, double, float, char і т. д. А ключове слово class повідомляє компілятору, що в шаблоні функції як параметр будуть використовуватися користувацькі типи даних, тобто класи. Але це правило не стало обов’язковим навіть на рівні стандартів стилю.

Оголошення шаблону класу дає наступні переваги:

•уникається повторюваність програмного коду для різних типів даних. Програмний код (методи, функції) пишеться для деякого узагальненого типу T. Назва узагальненого типу можна давати будь-яку, наприклад, TTT;

•зменшення текстової частини програмного коду, і, як наслідок, підвищення читабельності програм;

•забезпечення зручного механізму передачі аргументів в шаблоні класу з метою їх обробки методами класу.

В прикладі декларується шаблон класу, що містить методи, які виконують наступні операції над деяким числом:

•множення числа на 2;

•ділення одного числа на інше. Для цілих типів виконується ділення націло;

•возведення числа в квадрат (ступень 2).

Декларація шаблону класу має вигляд:

```
// шаблон класу,
template <class T> class MyNumber{
    public:
    // конструктор
    MyNumber(void) { }

    // метод, множення на 2
    void Mult2(T* t);

    // метод, возведення в квадрат
    T MySquare(T x);

    // метод, ділення двох чисел типу T, результат - тип T
    T DivNumbers(T x, T y);
};
```

Для реалізації методів шаблону класу за межами класу потрібно вказувати template <class T1, ...class Tn> перед об'явою функцію.

```
// реалізація методу множення на 2
template <class T> void MyNumber<T>::Mult2(T* t){
    *t = (*t)*2;
}

// реализація методу, возведення в квадрат
template <class T> T MyNumber<T>::MySquare(T number){
    return (T)(number*number);
}

// метод ділення
template <class T> T MyNumber<T>::DivNumbers(T t1, T t2){
    return (T)(t1/t2);
}
```

Для використання шаблону в інший функції чи методі (зокрема в main()) потрібно об'явити тип використовувано класу обов'язково вказавши, яким типом ми інстанціюємо даний клас.

Примітка. Для параметрів шаблону класу так само як і для шаблонів функцій можна задавати параметри за замовченням.

Використання шаблону класу MyNumber в іншому програмному коді:

```
MyNumber <int> mi; // об'єкт mi класу для типу int
MyNumber <float> mf; // об'єкт mf класу для типу float
```

```
int d = 8;
float x = 9.3f;

// множення на 2
mi.Mult2(&d); // d = 16
mf.Mult2(&x); // x = 18.6

// возведення в квадрат
int dd;
dd = mi.MySquare(9); // dd = 81 - ціле число

double z;
z = mf.MySquare(1.1); // z = 1.21000... - дійсне

// ділення чисел
long int t;
float f;

t = mi.DivNumbers(5, 2); // t = 2 - ділення цілих чисел
f = mf.DivNumbers(5, 2); // f = 2.5 - ділення дійсних чисел
```

Загальна форма оголошення шаблону класу, що приймає аргументи

Бувають випадки, коли в шаблоні класу потрібно використовувати деякі аргументи. Ці аргументи можуть використовуватися методами, які описуються в шаблоні класу.

Загальна форма шаблону класу, що містить аргументи, наступна:

template <class T1, class T2, …, class Tn, *type1 var1,* type2 *var2, ...,* typeN *varN*> **class ClassName{**

// тіло шаблону класу

// ...

}

де

- T1, T2, …, Tn – назви узагальнених типів даних;
- *type1, type2, …, typeN* – конкретні типи аргументів з іменами *var1, var2, …, varN*;
- *var1, var2, …, varN* – імена аргументів, які використовуються в класі.

Примітка. В декларації шаблонів можна використовувати також інші шаблони та ініціалізувати за потреби й аргументи стандартних і нестандартних типів за замовченням:

template <class T1, // параметр-тип
typename T2, // параметр-тип
int I, // параметр звичайного типу
T1 DefaultValue, // параметр звичайного типу
template <class> class T3, // параметр-шаблон
class C1 = char> // параметр за замовчуванням

Примітка 2. У версії стандарту C++11 була додана можливість використання шаблонів зі змінним числом параметрів.

Приклад: Давайте створимо шаблон класу Стек, де стек - структура даних, в якій зберігаються однотипні елементи даних. В стек можна додавати та видаляти дані. Новий елемент, який додається в стек, встановлюється в вершину стека. Видаляються елемент стека, який знаходиться на його вершині. У шаблоні класу Stack необхідно створити основні методи:

- push - додати елемент в стек;
- pop - видалити елемент з стека (на верхівці);
- top - вивести елемент що на верхівці стеку;
- printStack - виведення стеку на екран.

Отже, реалізуємо ці три методи, в результаті отримаємо найпростіший клас, який реалізує роботу структури стек. Також потрібні конструктори і деструктори.

```
#include <iostream>

template <class T, unsigned N=100>   // довжина стеку за замовченням
class Mystack {
    T a[N]; // будемо зберігати стек у масив
    unsigned num; // поточний розмір доданих даних

   public:
    Mystack(){ num=0; } // конструктор

    T top(){ // показати елемент
         return a[num];
    }

    bool push(T b){  // додати елемент

         if (num+1>=N) return false;
         a[num++]=b;
     return true;
```

```
    }


    T pop(){  //видалити елемент
         if (num==0) return 0; // тут краще робити виключення
(exception)
     return a[--num];
    }

   void printStack(){ // вивід змісту стеку
   for (int i=0;i<num;++i){
       std::cout<<a[i]<<",";
       std::cout<<std::endl;
   }

  }

};

// використання стеку
int main(){

   Mystack<int> s1; // стек цілих чисел
   Mystack<std::string> s2; // стек рядків

   s1.push(1);s1.push(2);
   s1.print();

   s2.push("aaa");s2.push("bbbb");
   s2.print();
}
```

Шаблони класів працюють точно так же, як і шаблони функцій: компілятор копіює шаблон класу, замінюючи типи параметрів шаблону класу на фактичні (передані) типи даних, а потім компілює цю копію. Якщо у вас є шаблон класу, але ви його не використовуєте, то компілятор не буде його навіть компілювати. Шаблони класів ідеально підходять для реалізації контейнерних класів, так як дуже часто таких класів доводиться працювати з різними типами даних, а шаблони дозволяють це організувати в мінімальній кількості коду. Хоча синтаксис дещо заплутаний, і повідомлення про помилки іноді можуть бути «об'ємними», шаблони класів є однією з важливіших конструкцій мови C++.

Спеціалізація шаблонів класу

Також для шаблонів-класів визначена можливість спеціалізації шаблонів. Прикладами спеціалізації шаблонів в C++ є:

- Реалізація функції sort() залежить від численних обмінів елементів значеннями. Якщо операція обміну значеннями є швидкою, як для атомарних типів чи , то її можна використовувати безпосередньо. Якщо ж вона є повільною, тоді потрібно створити для кожного елемента вказівник і здійснювати обмін значеннями серед вказівників.
- Стандартним прикладом спеціалізації шаблону є vector<bool> — спеціалізація шаблона послідовного контейнера бібліотеки STL, яка використовує однобітне зберігання значень типу bool.

Синтаксис спеціалізації шаблонів наступний:

```
#include <iostream>
using namespace std;
// шаблон класу - class template

template <class T> class mycontainer {
    T element;
  public:
    mycontainer (T arg) {
                  element=arg;
}
    T increase () {
      return ++element;
 }
};

template <>// !!! Увага: спеціалізація шаблону (template
specialization)
class mycontainer <char> {
    char element;
  public:
    mycontainer(char arg) {element=arg;}
    char uppercase () {
      if((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};
```

```
int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
}
```

Багатофайлове використання шаблонів

Шаблон не є ні класом, ні функцією - це трафарет, який використовується для створення класів або функцій. Таким чином, шаблони працюють не так, як звичайні функції або класи. У більшості випадків це не є проблемою, але на практиці трапляються різні ситуації. Працюючи зі звичайними класами ми поміщаємо визначення класу в заголовки, а визначення методів цього класу в окремий файл .cpp з аналогічним ім'ям. Таким чином, фактичне визначення класу компілюється як окремий файл всередині проекту. Однак з шаблонами все відбувається дещо інакше

```
// Array.h – інтерфейс шаблону
#ifndef ARRAY_H
#define ARRAY_H

#include <assert.h> // для assert()

template <class T>
class Array{
private:
    int m_length;
    T *m_data;

public:
    Array()    {
        m_length = 0;
        m_data = nullptr; // c++11 – порожній вказівник
    }

    Array(int length)    {
        m_data = new T[length];
        m_length = length;
    }

    ~Array()    {
```

```
        delete[] m_data;
    }

    void Erase()    {
        delete[] m_data;
        // Присвоюємо nullptr для m_data, щоб не отримати
незнищений вказівник!
        m_data = nullptr; // c++11
        m_length = 0;
    }


    T& operator[](int index)    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    // довжина масиву - ціла
    int getLength();
};

#endif

// Array.cpp – реалізація шаблону
#include "Array.h"

template <typename T>
int Array<T>::getLength() { return m_length; }

// ArrayDriver.cpp – файл для тестування шаблону
#include "Array.h"

int main(){
     Array<int> intArray(10);
     Array<double> doubleArray(10);

     for (int count = 0; count < intArray.getLength(); ++count){
          intArray[count] = count;
          doubleArray[count] = count + 0.5;
     }
```

```
    for (int count = intArray.getLength()-1; count >= 0; --
count)
        std::cout    <<    intArray[count]    <<    "\t"    <<
doubleArray[count] << '\n';

}
```

Вищенаведена програма зкомпілюється, але виникне таку помилку лінкеру: undefined reference to `Array <int> :: getLength ()

Дійсно, для використання шаблону компілятор повинен бачити як визначення шаблону (а не тільки оголошення), так і тип шаблону, що застосовується для створення екземпляра шаблону. Пам'ятаємо, що мова C++ компілює файли окремо. Коли заголовки Array.h підключаються в main.cpp, то визначення шаблону класу копіюється в цей файл. У ArrayDriver.cpp компілятор бачить, що нам потрібні два примірника шаблону класу: Array <int> і Array <double>, він створить їх, а потім скомпілює весь цей код як частину файлу ArrayDriver.cpp.

Однак, коли справа дійде до компіляції Array.cpp (окремим файлом), компілятор не буде пам’ятати, що ми використовували Array <int> і Array <double> в ArrayDriver.cpp і не створить екземпляр шаблону функції getLength (), який нам потрібен для виконання програми. Ми отримаємо помилку лінкера, так як компілятор не зможе знайти визначення Array <int> :: getLength () або Array <double> :: getLength ().

Цю проблему можна вирішити кількома способами.

Найпростіший варіант - помістити код з Array.cpp в Array.h нижче класу. Таким чином, коли ми будемо підключати Array.h, весь код шаблону класу (повне оголошення і визначення як класу, так і його методів) буде знаходитися в одному місці. Плюс цього способу - простота. Мінус - якщо шаблон класу використовується в багатьох місцях, то ми отримаємо багато локальних копій шаблону класу, що збільшить час компіляції і лінковки файлів (лінкер повинен буде видалити дублювання визначень класу і методів, щоб виконуваний файл не був «занадто роздутим»). Рекомендується використовувати це рішення до тих пір, поки час компіляції або лінковки не є проблемою. Якщо ви вважаєте, що розміщення коду з Array.cpp в Array.h зробить Array.h занадто великим, то альтернативою буде перейменування Array.cpp в Array.inl (.inl від англ. «Inline» = «вбудований»), а потім підключення Array.inl з нижньої частини файлу Array.h. Це дасть той же результат, що і розміщення всього коду в заголовки, але таким чином код вийде трохи чистіше. Є ще рішення - підключення файлів .cpp, але цей варіант не рекомендується використовувати через нестандартного застосування директиви #include.

Найбільш розповсюджений альтернативний варіант - використовувати підхід трьох файлів: визначення шаблону класу зберігається в заголовку, а визначення методів шаблону класу зберігаються в окремому файлі .cpp. Потім

додається третій файл, який містить всі необхідні нам екземпляри шаблону класу.

Наприклад, файл ArrayTemplates.cpp:

```
// Ми гарантуємо, що компілятор побачить повне визначення шаблона класу Array
#include "Array.h"
#include "Array.cpp" // ми трохи порушуємо правила включень C++, але лише в одному місці

// Тут ми за допомогою #include підключаємо всі .h и .cpp з визначеннями шаблонів що нам потрібні
 template class Array<int>; // створюємо екземпляр шаблону класу Array<int>
template class Array<double>; // створюємо екземпляр шаблону класу Array<double>

// створюємо явно інші екземпляри шаблонів класу, які потрібні
```

Після цього компілюємо разом проект, наприклад:

```
g++ Array.cpp ArrayDriver.cpp ArrayTemplates.cpp -std=c++11
```

Частина ArrayTemplates змусить компілятор явно створити зазначені екземпляри шаблону класу. У прикладі, наведеному вище, компілятор створить Array <int> і Array <double> всередині ArrayTemplates.cpp. Оскільки ArrayTemplates.cpp знаходиться всередині нашого проекту, то він зкомпілюється і вдало зв'яжеться з іншими файлами (пройде лінкінг). Цей метод більш ефективний, але вимагає створення / підтримки третього файлу (ArrayTemplates .cpp) для кожної з програм (проектів) окремо.

Наслідування класів шаблонів

Іноді потрібно використовувати наслідування для шаблонів класів, які використовуються в програмі.

Наприклад, нехай ми маємо шаблон класу Point2D, який має член x y, та методи для встановлення та виведення цих членів:

```
template<class T> class Point2D{
   T x;
   T y;
T getX();
T getY();
void setX(T);
void setY(T);
```

```
};
```

Тобто тепер можна визначити об’єкти Point2D визначеного типу, наприклад, Point2D<int>.

Зауважимо, що Point2D - це не клас шаблону, а шаблон класу. Тобто це шаблон, з якого можна створювати класи. Point2D <int> - це такий клас (це не об'єкт, але, звичайно, ви можете створити об'єкт з цього класу так само, як ви можете створити об'єкти з будь-якого іншого класу). Іншим таким класом буде Point2D <char>. Зверніть увагу, що це абсолютно різні класи, які не мають нічого спільного, крім того, що вони були створені з одного і того ж шаблону класу.

Нехай потрібно створити клас Point3D, який успадковує клас Point2D. Оскільки Point3D сам по собі не є шаблоном, то не можна ввести Point3D <int>. Оскільки Point2D не є класом, то безпосередньо з нього не можна вивести клас Point3D. Тобто при означенні Point3D необхідно вказати, з якого класу буде відбуватися спадкування. Це вірно незалежно від того, створюються ці класи з шаблону чи ні. Два об'єкти одного класу просто не можуть мати різні ієрархії успадкування.

Але можна наслідувати клас від інстанційованого класу Point2D (або декількох з них). Наприклад, оскільки Point2D <int> це клас, то можна вивести з нього Point3D:

```
 class Point3D:  public Point2D<int>{
  int z;
  // ...
};
```

Оскільки Point2D<int> та Point2D<char> - це різні класи, ви навіть можете отримати похідні від обох одночасно (проте при доступі до їх членів вам доведеться мати справу з неоднозначностями):

```
class PointX:
  public Point2D<int>,
  public Point2D<char>
{
  // ...
};
```

Тепер у мене є клас Point3D, який успадковує клас Point2D. Оскільки Point3D сам по собі не є шаблоном, не можна ввести Point3D <int>.

Для того, щоб створити клас наслідник шаблоном, потрібно зробити наступне:

```
template<typename T> class Point3D:
  public Point2D<T>{
  // ...
};
```

Тепер у нас є шаблон Point3D, з якого можна отримати клас Point3D <int>, похідний від Point2D <int>, і інший клас Point3D <char>, похідний від Point2D <char>. Можливо, було б непогано мати один тип Point3D, щоб ви могли передавати всі види Point3D однієї і тій самій функції (яка сама по собі не повинна знати тип області). Оскільки класи Point3D <T>, створені шляхом створення екземпляру шаблону Point3D, формально незалежні один від одного, той варіант, що був зроблений не працює таким чином. Однак тут можна використовувати множинне наслідування:

```
class Point3D // не є наслідником Point2D
{
  // Незалежний інтерфейс
};

template<typename T> class SpecificPoint3D:
  public Point3D,
  public Point2D<T>
{
  // Залежний інтерфейс
};

void foo(Point3D&); //Функція що працює з загальним типом
Point3D

int main(){
  SpecificPoint3D <int> int_pt;
  foo(int_pt);

  SpecificPoint3D <char> char_pt;
  foo(char_pt);
}
```

Якщо важливо, що універсальний Point3D є похідним від універсального Point2D, можна зробити той самий трюк і з Point2D:

```
class  Point2D{
  // загальний інтерфейс Point2D
};

class Point3D:
  public  virtual  Point2D{  //  virtual  тому  що  "diamond
inheritance"

  // інтерфейс для  Point3D
};

template<typename T> class SpecificPoint2D :
```

```
  public virtual  Point2D{
  // реалізація Point2D для типу T
};

template<typename T> class SpecificPoint3D:
  public Point3D, /* можливо це також буде virtual, для
подальшого наслідування*/
  public SpecificPoint2D <T> /* тут не потрібно віртуальне
наслідування*/
{
  // реалізація Point3D для типу T
};
```

Стандартна бібліотека шаблонів STL

Потреба в бібліотеці шаблонів

Створення шаблонів класів дозволяє програмістам створювати корисні програмні та алгоритмічні класи та функції. Звичайно, розробники Сі++ скористалися цією можливістю для того щоб збільшити функціонал та можливості мови. В доповнення до бібліотеки потокового введення/виведення та бібліотеки роботи з рядками <string> було створено потужну систему бібліотек побудованих за об'єктно-орієнтовним принципом та об'єднаних спільною ідеєю та структурою. Ця бібліотека робить легкою можливість використання багатьох відомих алгоритмів на Сі++. Вона була розроблена спочатку як вільна бібліотека, а потім була включена в стандарт та отримала назву "стандартна бібліотека шаблонів" (STL – Standard Template Library).

Огляд стандартної бібліотеки шаблонів C ++ (STL)

Стандартна бібліотека шаблонів (STL) - це набір шаблонів C ++ для створення загальних структурних даних і функцій, таких як списки, стеки, масиви і т.д. Це бібліотека контейнерних класів, алгоритмів і ітераторів. Це узагальнена бібліотека, тому її параметри параметризуються. Робоче знання шаблонів класів є обов'язковою умовою для роботи з STL.
Стандартна бібліотека шаблонів складається з таких чотирьох компонентів:

- Algorithms (алгоритми): набір корисних функцій, спеціально призначених для використання на діапазонах елементів, які містяться в заголовних файлах <algorithm> та <numeric>. Вони діють на контейнери або масиви і забезпечують засоби для виконання корисних дій з вмістом контейнерів.
- Containers (контейнери): шаблони класів, що дозволяють зберігати об'єкти

і дані різних типів за допомогою різних структур даних, а отже дозволяють оптимізувати роботу з ними під різні типи задач.

- Functions (функціонали або функтори): STL включає класи, які перевантажують оператор виклику функції. Екземпляри таких класів називаються функціональними об'єктами або функторами. Функтори дозволяють налаштувати роботу пов'язаної функції за допомогою параметрів, що передаються.
- Iterators (ітератори): STL включає класи, що можуть використовуватися для роботи з послідовністю значень в різних контейнерах забезпечуючи таким чином універсальність типів даних STL.
- Утілити та псевдоконтейнери: ще одним компонентом, що може включатись до STL – це утілити бібліотеки C++ та так звані псевдоконтейнери, зокрема ті, що входять до складу пакетів <utility>, <bitset>, <tuple>, <valarray> тощо, а іноді сюди відносять також і бібліотеку <string>.

Утілити utility

Розгляд бібліотеки ми почнемо з розгляду деяких корисних функцій та класів бібліотеки <utility>.

Контейнер Пара

До C++11 в даній бібліотеці містився лише шаблон класу Пари (pair).
Контейнер Pair (пара) являє собою простий контейнер, визначений у заголовному файлі <utility>, що складається з двох елементів даних або об'єктів. Даний шаблон класу можна описати наступним чином:
• Перший елемент позначається як «first», а другий - як «second», а порядок фіксований (first, second).
• Пара використовується для об'єднання двох значень, які можуть бути різними за типом. Пара забезпечує спосіб зберігання двох неоднорідних об'єктів у вигляді однієї змінної.
• Пара може бути означена, скопійована та порівняна. Масив об'єктів, виділених для відображення (map) або hash_map, за замовчуванням мають тип "пари", в якому всі елементи "first" є унікальними ключами, пов'язаними зі значеннями "second".
• Для доступу до елементів використовується ім'я змінної, за яким слідує оператор точки, за яким слідує ключове слово first або second.

```
//C++ програма - демонстрація пари (pair) STL
#include <iostream>
#include <utility>
```

```
int main() {
std::pair <int, char> PAIR1;
PAIR1.first = 100;
PAIR1.second = 'G';

std::cout << PAIR1.first << " ";
std::cout << PAIR1.second << "\n" ;
std::pair <std::string,unsigned> PAIR2 ("Мехмат", 65);

std::cout << PAIR2.first << " ";
std::cout << PAIR2.second << "\n" ;
}
```

Результат:

```
100 G
Мехмат 65
```

Ініціалізація пари:

pair (data_type1, data_type2) Pair_name (value1, value2) ;

Приклади:

```
pair g1; //за замовченням
pair g2(1, 'a'); //ініціалізоване, різні типи у пари
pair g3(1, 10); // ініціалізоване, однаковий тип у пари
pair g4(g3); //копія g3
```

Ще один варіант ініціалізації — визначена в цьому ж пакеті функція make_pair(), яка визначена наступним чином:

```
template <class T1,class T2>
pair<T1,T2> make_pair (T1 x, T2 y){
return ( pair<T1,T2>(x,y) );
}
g2 = make_pair(1, 'a');
```

В класі визначені конструктори:

- Стандартний конструктор за замовченням:

pair();

- Стандартний конструктор копіювання:

template<class U, class V> pair (const pair<U,V>& pr);

- Конструктор ініціалізації (сеттер):

pair (const first_type& a, const second_type& b);

Примітка: Якщо поле не ініціалізоване конструктором за замовченням, то воно автоматично ініціалізується нулем.

```
using namespace std;
```

```
int main() {
    pair <int, double> PAIR3 ;
    pair <string, char> PAIR4 ;

    cout << PAIR3.first ;  //ініціалізовано 0
    cout << PAIR3.second ; //ініціалізован 0
    cout << " ";

    cout << PAIR4.first ;  //виводить NULL
    cout << PAIR4.second ; //виводить NULL
}
```

Методи класу

make_pair() : створює пару без визначення типів полів явно

Pair_name = make_pair (value1,value2);

Приклад

```
pair <int, char> PAIR1 ;
pair <string, double> PAIR2 ("Студент", 4.23) ;
pair <string, double> PAIR3 ;

PAIR1.first = 100;
PAIR1.second = 'G';

PAIR3 = make_pair ("Мехмат кращій",4.56);

cout << PAIR1.first << " ";
cout << PAIR1.second << endl ;

cout << PAIR2.first << " ";
cout << PAIR2.second << endl ;

cout << PAIR3.first << " ";
cout << PAIR3.second << endl ;
```

Оператори **(=, ==, !=, >=, <=) :**

- **(=)** Присвоює нову пару – конструктор копіювання.

pair& operator= (const pair& pr);

- **Рівність (==) – п**орівнює поля pair1 та pair2. Пари рівні якщо pair1.first рівне pair2.first та pair1.second рівне pair2.second.
- **Нерівність (!=) – п**ротилежний до рівності.
- **Логічні (>=, <=) оператори порівняння - п**орівнює лише перше поле пари.

Приклад.

```
#include <iostream>
#include <utility>
using namespace std;

int main(){
  pair<int, int>pair1 = make_pair(1, 12);
  pair<int, int>pair2 = make_pair(9, 12);
  cout<<boolalpha;
  cout << (pair1 == pair2) << endl;
  cout << (pair1 != pair2) << endl;
  cout << (pair1 >= pair2) << endl;
  cout << (pair1 <= pair2) << endl;
  cout << (pair1 > pair2) << endl;
  cout << (pair1 < pair2) << endl;
}
```

Результат:

```
false
true
false
true
false
true
```

Примітка 1: З C++20 у якості операторів порівняння радять використовувати оператор трьохстороннього порівняння (operator<=>)

Примітка 2: з C++11 додано також метод **swap, що зм**інює місцями значення однієї пари зі значеннями іншої пари тих самих типів.

//pair STL

```
#include <iostream>
#include <utility>

int main(){
  std::pair<char, int>pair1 = std::make_pair('A', 1);
  std::pair<char, int>pair2 = std::make_pair('B', 2);
  std::cout << "Before swap:\n ";
  std::cout << "Pair1 = "<< pair1.first << " "<< pair1.second
<<"\t" ;
  std::cout << "Pair2 = "<< pair2.first << " "<< pair2.second;
  pair1.swap(pair2);

  std::cout << "\nAfter swapping:\n ";
```

```
  std::cout << "Pair1 = "<< pair1.first << " "<< pair1.second
<<"\t" ;
  std::cout << "Pair2 = "<< pair2.first << " "<< pair2.second;
}
```

Результат:

```
Before swap:
 Pair1 = A 1  Pair2 = B 2
After swapping:
 Pair1 = B 2  Pair2 = A 1
```

З використанням цього шаблону полегшується використання в багатьох ситуаціях роботи зі стандартними контейнерами STL, особливо це стосується шаблонів класів відображення (хештаблиць) map/multimap.

Контейнери

Контейнери або класи контейнерів зберігають об'єкти і дані. В загальній складності сім стандартних класів контейнерів першого класу і три класи адаптерів контейнерів і лише сім файлів заголовків, які забезпечують доступ до цих контейнерів або адаптерів контейнерів.

• **Контейнери послідовності** (**Sequence containers**) - реалізують структури даних, до яких можна звертатися послідовно тобто за номером елементу:

- vector
- list
- deque
- arrays (C++11)
- forward_list(C++11)

• **Контейнери адаптери (Container Adaptors)** - реалізують інтерфейс для контейнерів баз даних, найбільш прості структури даних, що дозволяють додавати та видаляти елементи без забезпечення послідовного доступу:

- queue
- priority_queue
- stack

• **Асоціативні контейнери (Associative Containers)** – структури даних що зберігають сортовані структури даних, що дозволяють швидкий пошук (зі складністю O(log n)):

- set
- multiset
- map
- multimap

• З С++11 до стандартних контейнерів додали також **невідсортовані асоціативні контейнери (Unordered Associative Containers)** які складаються з невідсортованих структур даних- хеш таблиць та відображень:

- unordered_set (C++11)
- unordered_multiset (C++11)
- unordered_map (C++11)
- unordered_multimap (C++11)

Контейнери адаптори

Стек (Stack)

Стек – це шаблон класу який реалізує концепцію доступу до даних LIFO (Last In First Out – останній зайшов, перший вийшов), в який елемент додається до верхівки структури та відповідно опускає вниз той елемент, що до цього знаходився на верхівці стеку. Коли ж відбувається операція видалення елементу – видаляється верхівка стеку, та той елемент, що знаходився нижче (якщо він був) повертається на верхівку. Нижче нової вершини так саме знаходиться попередник нового елементу вершини і так далі.
Одна з можливих реалізацій шаблону була розглянута в попередньому розділі. В стандартній бібліотеці, зрозуміло, більш якісна версія класу шаблону, що дозволяє виділяти пам'ять під кожен новий елемент та кидає виключення у випадку некоректного видалення елементу.
Для використання цього контейнеру потрібно *підключити заголовний файл <stack>*.
Методи цього класу стеку працюють дуже швидко (час виконання – O(1)), а саме:

- empty() – булева функція, що повертає true, якщо стек порожній;
- size() – повертає розмір стеку;
- top() – повертає вказівник на останній елемент(верхівку) стеку;
- push(T g) –додає елемент 'g' у верхівку стеку;
- pop() – видаляє верхівку стеку.

Також для стеку визначені оператори присвоєння та порівняння (>,>=,==,!=,<,<=) які здійснюють відповідні поелементні порівняння для вмістів двох стеків.
Конструктори визначений за замовченням та аналог копіконструктору, що приймає стек.
Примітка. З С++11 додано також методи swap() для обміну значень двох стеків та emplace() - синонім push(), який дозволяє додавати елементи в колекцію по аргументах типу, тобто без зайвого копіювання та переміщення у рамках move-семантики стандарту С++11.

```
// C++ програма для демонстрації STL стеку
#include <iostream>
```

```
#include <string>
#include <stack>
using namespace std;
// виводить стек у зворотньому порядку
void showstack(const stack <string> & s){
stack <string> copy_s(s); // copy-constructor to create copy of
s
// поки стек не порожній – виводить верхівку та видаляє її
while (!copy_s.empty()){
cout<<'\t'<<copy_s.top();
copy_s.pop();
}
cout << '\n';
}

int main(){
stack <string> s;
s.push("sator");
s.push("arepo");
s.push("tenet");
s.push("opera");
s.push("rotas");
cout <<"The stack is : ";
showstack(s);
//The stack is : rotas opera tenet arepo sator
cout<<"\ns.size():"<<s.size();//5
cout << "\ns.top() : " << s.top();//rotas
cout << "\ns.pop() : "; //
s.pop();
showstack(s); // opera tenet arepo sator
//s.pop() : // tenet arepo sator
}
```

Результат:

```
The stack is :     rotas     opera tenet arepo sator
s.size():5
s.top() : rotas
s.pop() :     opera     tenet arepo sator
```

Черга (Queue) в (STL)

На відміну від стеку, черга - шаблон класу який реалізує концепцію доступу до даних FIFO (First In First Out – перший зайшов, перший вийшов), в який елемент додається до кінця структури та видаляється елемент, що стоїть в початку структури. Тут пряма аналогія з “чергою” у магазині.
Для використання цього контейнеру потрібно підключити заголовний файл <queue>.
Методи класу працюють зі складністю O(1):

- empty() – булева функція, що повертає true, якщо черга порожня та false в іншому випадку;
- size() – повертає розмір черги;
- push(T g) –додає елемент ‘g’ у кінець черги;
- pop() – видаляє початок черги;
- front() - повертає посилання на елемент початку черги;
- back() - повертає посилання на останній елемент черги.

Також для черги визначені оператори присвоєння та порівняння (>,>=,==,!=,<,<=) які здійснюють відповідні поелементні порівняння для вмісту двох черг.
Конструктори, що є для черги: конструктор за замовченням та аналог копіконструктору.

Примітка. З C++11 додано також методи swap() для обміну значень двох черг та emplace() - що має функціонал push().

```
// Queue in Standard Template Library (STL)
#include <iostream>
#include <queue>

using namespace std;
// виводить чергу в порядку заповнення для будь-якого стандартного типу
template <typename T>
void showq(const queue <T> & gq) {
// створюємо копію черги
queue <T> g = gq;
// видаляємо та дивимось всі елементи в неї з початку
while (!g.empty()) {
     cout << '\t' << g.front();
        g.pop();
 }
cout << '\n';
}
```

```
int main() {
   queue <int> q1;
   q1.push(10);
   q1.push(20);
   q1.push(30);
   q1.push(15);
   cout << "The queue q1 is : ";
   showq(q1);
   cout << "\nq1.size() : " << q1.size();
   cout << "\nq1.front() : " << q1.front();
   cout << "\nq1.back() : " << q1.back();
   cout << "\nq1.pop() : ";
   q1.pop();
   showq(q1);
}
```

Результат:

```
The queue q1 is :  10   20   30   15
q1.size() : 4
q1.front() : 10
q1.back() : 15
q1.pop() :    20   30   15
```

Пріоритетна черга (Priority Queue)в STL

Пріоритетна черга відрізняється від звичайної черги тим, що першим елементом на видалення йде не перший елемент, а елемент, що приймає найбільше значення. Тобто пріоритетна черга - це тип контейнеру-адаптеру, спеціально розроблений таким чином, що перший елемент черги є найбільшим із усіх елементів черги, а елементи контейнеру розташовані в порядку неспадання (отже, ми можемо бачити, що кожен елемент черги має пріоритет (фіксований порядок).

Для використання цього контейнеру потрібно підключити заголовний файл <queue>.

Методи контейнеру:

- empty() – булева функція, що повертає true, якщо черга порожня. Час виконання O(1);
- size() – повертає розмір черги – час виконання O(1);
- top() – повертає вказівник на останній елемент(верхівку) черги. Час

виконання O(1);

- push(T g) –додає елемент ‘g’ у кінець черги. Час виконання повинен бути O(log n) ;
- pop() – видаляє початок черги. Час виконання час повинен бути O(log n).

Також для пріоритетної черги визначені оператори присвоєння та порівняння (>,>=,==,!=,<,<=) які здійснюють відповідні поелементні порівняння для вмісту двох черг.

Конструктори для цього класу - конструктор за замовченням та аналог копіконструктору, що приймає цю колекцію.

```
// Queue in Standard Template Library (STL)
#include <iostream>
#include <queue>

using namespace std;
// виводить чергу в порядку пріоритету для стандартних типів
template<class T>
void showpq(const priority_queue <T> & gq) {
  priority_queue <T> g(gq);
  while (!g.empty()){
   cout << '\t' << g.top();
   g.pop();
   }
}
// виводить чергу в порядку пріоритету для типу Пара від двох цілих – спеціалізація  showpq
template<>
void showpq(const priority_queue <pair<int,int> > & gq) {
  //‘>>’ should be ‘> >’ within a nested template argument list before C++11
  priority_queue <pair<int,int> > g(gq);
  while (!g.empty()){
   pair<int,int> tmp = g.top();
   cout << "\t(" << tmp.first<<", "<<tmp.second<<")";
   g.pop();
   }
}
int main() {
  priority_queue <int> q2;
   q2.push(10);
   q2.push(30);
```

```
   q2.push(20);
   q2.push(5);
   q2.push(1);
   cout << "The priority queue q2 is : ";
   showpq(q2);
   cout << "\nq2.size() : " << q2.size();
   cout << "\nq2.top() : " << q2.top();
   cout << "\nq2.pop() : ";
   q2.pop();
   showpq(q2);

   priority_queue <pair<int,int> > q3;/* коректно до c++20 – інакше для pair не визначено оператор < */
   q3.push(make_pair(1,2));
   q3.push(pair<int,int>(3,4));
   q3.push(make_pair(1,4));
   q3.push(make_pair(2,10));
   cout << "The priority queue q3 is : ";
   showpq(q3);
}
```

Результат:

```
The priority queue q2 is :  30   20   10   5    1
q2.size() : 5
q2.top() : 30
q2.pop() :    20   10   5    1
The priority queue q3 is :  (3, 4)    (2, 10) (1, 4) (1, 2)
```

Примітка. Звертаємо увагу, що тут використано контейнер від шаблону класу. До C++11 потрібно було завжди в цьому випадку писати визначення типу вигляду:
priority_queue <pair<int,int> > q3; // обов’язковий пробіл між двома останніми дужками > >

І хоча останні стандарти дозволяють не робити цей пробіл, для сумісності коду краще цей пробіл ставити.

Контейнери послідовності

Контейнери послідовності (Sequential containers) або контейнери послідовного доступу дозволяють зберігати масиви даних таким чином, щоб можна було б, на відміну від контейнерів адаптерів, отримати доступ до довільного елементу

контейнеру. Таким чином, в цих контейнерах можна проходити зміст контейнеру за допомогою аналогу вказівника – ітератору. Різниця між даними контейнерами полягає в сутності структур даних та можливостях контейнера (дек – розширений функціонал для контейнерів адаптерах, вектор та масив – для роботи з динамічним та сталого розміру масивами, списки – для даних де потрібно часто додавати/видаляти дані всередині).

Дек (Deque)

Дек або двонаправлена черга є контейнером послідовності з функцією розширення і стиснення на обох кінцях. Це фактично об'єднання структур стек та черга з доданим функціоналом послідовного доступу. Вони схожі на вектори, але більш ефективні у випадку вставки і видалення елементів. На відміну від векторів, для деків послідовне виділення пам'яті не може бути гарантованим. Деки в основному є реалізацією структури даних двонаправлена черга. Структура даних черг дозволяє вставляти тільки в кінці і видаляти з початку. Це схоже на чергу в реальному житті, де люди видаляються з початку черги і додаються назад. Можна сказати, шо дек - це особливий випадок черг, де операції вставки та видалення можливі на обох кінцях.
Методи для цієї структури наступні:

- insert() – вставляє в дек нові елементи та повертає ітератор на початок встановлюваних елементів;
- max_size() – максимальний розмір контейнера;
- assign() – присвоює нові значення контейнеру;
- resize() – змінює розмір деку;
- push_front() – вставляє елементи на початок;
- push_back() – вставляє елемент в кінець;
- pop_front() – видаляє елемент в початок
- pop_back() – видаляє елемент з кінця;
- front() – повертає посилання на перший елемент;
- back() – повертає вказівник на останній елемент;
- clear() – видаляє всі елементи деку;
- erase() – видаляє елементи всередині вказаного діапазону;
- empty() – перевіряє чи порожній дек;
- size() – повертає розмір деку;
- operator[] – повертає елемент заданої позиції;
- at() – повертає елемент заданої позиції;
- operator= – присвоює новий дек.

Також для черги визначені оператори присвоєння та порівняння (>,>=,==,!=,<,<=) які здійснюють спочатку порівняння для розмірів контейнеру, а потім відповідні поелементні порівняння для вмістів двох контейнерів.
Крім того, визначені методи для роботи з ітераторами, які будуть розглянуті в розділі про ітератори.

Шаблон класу дек має також наступні конструктори:

1) Конструктор за замовченням – створює порожній дек.
2) Конструктор заповнення - ініціалізує контейнер з даною кількістю даних елементів:

explicit deque (size_type n, const value_type& val = value_type())// Будує контейнер з *n*елементів. Кожен елемент дорівнює *val*.

Приклад:

deque d1(5, 10); // Вміст деку d1 : 10,10,10,10,10

3) Копіконструктор — копіює вміст іншого деку.
4) Конструктор за інтервалом (range constructor) — конструює дек за двома ітераторами іншого деку.

Примітка. З C++11 додано також методи swap() для обміну значень двох деків та методи додавання елементів emplace_front(), emplace_back(), а також метод для зміни розміру структури під дані shrink_to_fit(). Також було додано move-конструктори та конструктори за списком ініціалізації.

Приклад:

```
#include <iostream>
#include <deque>
#include <stack>
using namespace std;

// виведення деку від стандартного класу
template<typename T>
void showdq(const deque <T> & g){
 for(size_t i=0; i<g.size();++i)
    cout << '\t'<< g[i];
 cout << '\n';
}
// виведення стеку
template<class T>
void printStack(const stack<T> & st){
 stack<T> st1(st);
 while(!st1.empty()){
    cout << ", "<< st1.top();
```

```
    st1.pop();
 }
 cout << ";\t";
}
// виведення деку від стеку
template<typename T>
void showdq(const deque<stack<T> > & g){
  for(size_t i=0; i<g.size();++i) {
    cout << "stack "<< i<<":";
    printStack(g[i]);
 }
 cout << '\n';
}

int main(){
// дек від рядків Ci
deque <char*> deq1;
deq1.push_back((char*)"cadabra");
deq1.push_front((char*)"abra");
deq1.push_back((char*)"bums");
deq1.push_front((char*)"\n");
cout << "The deque deq1 is : ";
showdq(deq1);
cout << "\ndeq1.size() : "<< deq1.size();
cout << "\ndeq1.max_size() : "<< deq1.max_size();
cout << "\ndeq1.at(2) : "<< deq1.at(2);
cout << "\ndeq1.front() : "<< deq1.front();
cout << "\ndeq1.back() : "<< deq1.back();
cout << "\ndeq1.pop_front() : ";
deq1.pop_front();
showdq(deq1);
cout << "\ndeq1.pop_back() : ";
deq1.pop_back();
showdq(deq1);
// дек від стеків
deque<stack<int> > deq2;
stack<int> a1;
a1.push(1);a1.push(2);
deq2.push_back(a1);

stack<int> a2(a1);
a2.push(3);
```

```
deq2.push_back(a2);

deq2.push_front(a1);
a1.pop();
deq2.push_back(a1);
showdq(deq2);
}
```

Результат:

```
The deque deq1 is :
     abra cadabra   bums

deq1.size() : 4
deq1.max_size() : 2305843009213693951
deq1.at(2) : cadabra
deq1.front() :

deq1.back() : bums
deq1.pop_front() :      abra cadabra   bums

deq1.pop_back() :  abra cadabra
stack 0:, 2, 1;    stack 1:, 2, 1;    stack 2:, 3, 2, 1; stack
3:, 1;
```

Вектор

Вектор є, мабуть, найбільш популярним та застосовуваним контейнером STL. Вектори є реалізацією динамічних масивів з можливістю автоматично змінювати розмір, коли елемент вставляється або видаляється, а їх зберігання автоматично обробляється контейнером. Елементи вектору розміщуються в контейнері, так що до них можна отримати доступ і пройти всі елементи за допомогою ітераторів. У векторах дані вставляються в кінець. Вставка в кінець займає змінний час, оскільки іноді може виникнути потреба у розширенні масиву. Видалення останнього елемента займає лише сталий час (O(1)), оскільки зміна розміру не відбувається. Вставка та видалення елементу на початку або всередині є лінійними по часу O(n).

Методи, які містить клас вектор можна класифікувати наступним чином:

- **Модифікатори даних (Modifiers);**
- **Ідентифікатори та модифікатори обсягу (Capacity);**
- **Методи доступу до елементів (Access);**

- **Робота з ітераторами (Iterators).**

До С++11 шаблон класу множина мав наступні конструктори
1) конструктор за замовченням (default constructor), що створює порожній вектор
2) Копіконструктор — копіює вміст іншого вектору
3) Конструктор за інтервалом (range constructor) — конструює вектор за двома ітераторами іншої колекції.
4) Конструктор заповнення - ініціалізує контейнер з даною кількістю даних елементів:
explicit deque (size_type n, const value_type& val = value_type())// Будує контейнер з *n* елементів. Кожен елемент дорівнює *val*.

Приклад:
vector v1(3, 'A'); // Вміст v1 : A,A,A

Модіфікатори даних (Modifiers):

- assign(n, v) – присвоюємо (ініціалізуємо) значення у векторі — n значень v присвоюється у вектор;
- push_back(g) – додає елемент g у кінець вектору;
- pop_back() – видаляє елемент з кінкця вектору;
- insert(g, it_beg) – додає елемент(елементи) у вказану ітератором позицію;
- erase(it_begin, it_end) – видаляє елементи з вектору по вказаному ітераторами інтервалу;
- clear() – видаляє всі елементи з вектору;
- emplace(g) – розширює вектор вставляючи нові елементи на позицію вказану ітератором (С++11);
- emplace_back(g) – додає нові елементи в кінець вектору(С++11);
- swap(v) – міняє значення векторів з одного в інший. Розміри векторів (але не тип) можуть відрізнятись(С++11).

```
// C++ program to illustrate the methods Modifiers in vector
#include <iostream>
#include <vector>
using namespace std;

int main(){
// Assign vector
vector<int> v;
// fill the array with 10 five times
v.assign(5, 10);
```

```
cout << "The vector elements are: ";
for(int i = 0; i < v.size(); i++)
cout << v[i] << " ";

// inserts 15 to the last position
v.push_back(15);
int n = v.size();
cout << "\nThe last element is: "<< v[n - 1];

// removes last element
v.pop_back();

for(int i = 0; i < v.size(); i++)
 cout << v.at(i) << " ";
// erases the vector
v.clear();
cout << "\nVector size after erase(): "<< v.size();


// prints the vector
cout << "\nThe vector elements are: ";
for(int i = 0; i < v.size(); i++)
 cout << v.at(i) << " ";

// inserts 5 at the beginning
v.insert(v.begin(), 5);

cout << "\nThe first element is: "<< v[0];

// removes the first element
v.erase(v.begin());

cout << "\nThe first element is: "<< v[0];
}
```

Результат:

```
The vector elements are: 10 10 10 10 10
The last element is: 1510 10 10 10 10
Vector size after erase(): 0
The vector elements are:
The first element is: 5
The first element is: 5Size : 5
```

Ідентифікатори та модифікатори обсягу (Capacity)

- size() – кількість елементів вектору;
- max_size() – максимальна кількість елементів вектору;
- capacity() – розмір алокатору, виділеного під цій контейнер, тобто справжній розмір даного вектору;
- resize(– змінює розмір вектору на заданий;
- empty() – Повертає true, якщо контейнер порожній;
- reserve(–виділяє пам'ять для зберігання рівно n елементів;
- shrink_to_fit() – Зменшує розмір контейнеру видаляючи неініціалізовані елементи (C++11).

Приклад:

```
vector<int> g1;
for(int i = 1; i <= 5; i++)
  g1.push_back(i);

cout << "Size : "<< g1.size();
cout << "\nCapacity : "<< g1.capacity();
cout << "\nMax_Size : "<< g1.max_size();

// resizes the vector size to 4
g1.resize(4);

// prints the vector size after resize()
cout << "\nSize : "<< g1.size();

// checks if the vector is empty or not
if(g1.empty() == false)
cout << "\nVector is not empty";
else
cout << "\nVector is empty";

for(vector<int>::iterator it = g1.begin(); it != g1.end(); it++)
  cout << *it << " ";
```

Результат:

```
Capacity : 8
Max_Size : 4611686018427387903
Size : 4
Vector is not empty1 2 3 4
```

Методи доступу до елементів:

- at(n) – повертає посилання на ‘n’-ий елемент вектору;
- operator [n] – перевантажений оператор для доступу до ‘n’-го елементу вектору;
- front() – повертає посилання на перший елемент контейнеру;
- back() – повертає посилання на останній елемент контейнеру;
- data() – повертає вказівник на місце де зберігаються дані.

Приклад:

```
string massiv[] = {"first", "second", "third", "forth",
"fifth"};
vector<string> g2(massiv, massiv + 5);

cout << "\nReference operator [g] : g1[2] = "<< g2[2];
cout << "\nat : g1.at(4) = "<< g2.at(4);
cout << "\nfront() : g1.front() = "<< g2.front();
cout << "\nback() : g1.back() = "<< g2.back();
// pointer to the first element
string* pos = g2.data();
cout << "\nThe first element is "<< *pos;
```

Результат:

```
Reference operator [g] : g1[2] = third
at : g1.at(4) = fifth
front() : g1.front() = first
back() : g1.back() = fifth
The first element is first
```

Різницю між оператором [] та методом at() полягає в обробці ситуації з невірним індексом вектору. При використанні функції at() при спробі звернення по неприпустимому індексу буде генеруватись виключення out_of_range, в той яас як при використанні квадратних дужок поведінка компілятора невизначена:

```
    int mas[] = { 1, 2, 3, 4, 5};
    std::vector<int> numbers(mas,mas+5);
    try    {
        int n = numbers.at(5); //oops...
        std::cout<<"n="<<n;
    }
    catch (std::out_of_range e)    {
        std::cout << "Caught: Incorrect index 1" << std::endl;
    }
```

```
    std::vector<int> numbers2(mas,mas+5);
    try    {
        int n = numbers2[5]; // ??? - поведінка невизначена
        std::cout<<"n="<<n; // n=0 або навіть n=502012792....
    }
    catch (std::out_of_range e)
    {
        std::cout << "Shouldnt caught: Incorrect index 2" <<
std::endl;
    }
```

Результат:

```
Incorrect index 1
n=502012792
```

Робота з ітераторами:

- begin() – повертає ітератор на перший об’єкт вектору
- end() – повертає ітератор на останній об’єкт вектору

Починаючі зі стандарту C++11 додано ще наступні варіанти доступу до ітераторів:

- rbegin() – повертає ітератор на останній об’єкт вектору як на початковий (reverse beginning). Рухається з останнього елементу до першого;
- rend() – повертає ітератор на перший об’єкт вектору як на останній (reverse beginning). Рухається він з останнього елементу до першого;
- cbegin() – повертає константний ітератор на перший елемент;
- cend() – повертає константний ітератор на останній елемент;
- crbegin() – повертає константний реверсивний оператор на початок;
- crend() – повертає константний реверсивний оператор на кінець вектору.

Список (list) у STL

Списки є контейнерами послідовностей, які дозволяють як і асоціативні контейнери не виділяти цілком однорідну пам'ять під масиви, а зберігати її “порціями” разом з посиланнями на попередній та наступний об’єкти. У порівнянні з вектором список має повільну обробку, але коли знайдена позиція, вставка і видалення є швидкими. Зазвичай, коли ми говоримо список, ми говоримо про подвійно пов'язаний список. Для реалізації однозв'язного списку потрібно використовувати forward_list (C++11).

- **Модифікатори даних (Modifiers)**

- **Ідентифікатори та модифікатори обсягу (Capacity)**
- **Методи доступу до елементів (Access)**
- **Робота з ітераторами (Iterators)**

Методи класу List:

Ідентифікатори та модифікатори обсягу:

- size() – повертає кількість елементів;
- resize()- **перевизначає розмір контейнеру;**
- empty() **– повертає чи порожній список (true) чи ні (false);**
- max_size()**–максимальна кількість елементів списку.**

Методи доступу до елементів (тут значно менше методів ніж у векторів):

- front() – перший елемент списку;
- back() **– останній елемент списку;**

Модифікатори даних (а ось ці методи значно різноманітні ніж у векторі):

- assign() **– визначає нові елементи шляхом заміни їх новими значеннями;**
- push_front(g) – додає елемент ‘g’ у початок списку;
- push_back(g) – додає елемент ‘g’ в кінець списку;
- pop_front() – видаляє перший елемент списку, зменшує розмір списку на 1;
- pop_back() – видаляє останній елемент списку, зменшує розмір списку на одиницю;
- insert() – додає елемент перед поточним елементом списку;
- **clear() –** видаляє елементи списку, його розмір стає нульовим;
- erase() – видаляє елементи між двома вказаними вказівниками (ітераторами);
- merge() – зливає два списки в один;
- remove() – видаляє всі елементи списку, що є рівними даному;
- remove_if() – видаляє елементи, що задовольняють деякій булевій умові;
- reverse() – інвертує список;
- sort() – сортує елементи по неспаданню;
- splice()– передає елементи з одного списку в інший;
- unique() – видаляє дублюючи елементи списку.

Починаючи з C++11, додано наступні методи:

- emplace() – розширює список, додаючи в нього ще елементи;
- emplace_front() **–** вставляє елемент на початок списку;
- emplace_back() **–** вставляє елемент в кінець списку;

- swap()– обмінює значення двох списків, якщо в них співпадають розміри та тип.

Ітератори

- begin() – повертає ітератор що вказує на початок списку;
- end() – повертає ітератор, що вказує на кінець списку;
- rbegin() – повертає реверсивний ітератор на кінець списку;
- rend() – повертає реверсивний ітератор на початок;
- cbegin() – константний ітератор на початок;
- cend() – повертає константний ітератор на кінець;
- crbegin() – повертає константний реверсивний ітератор на початок;
- crend() – повертає константний оператор на кінець списку.

Приклад:

```
#include <iostream>
#include <list>
#include <iterator> // advance
using namespace std;

//function for printing the elements in a list
template <typename T>
void showlist(const list <T> & g){
   // to get iterator of template type - write template before
assignment
   typename list<T>::const_iterator it = g.begin();
   cout << *it;
   it++;
   for(; it != g.end(); ++it)
      cout << ','<< *it ;
  cout << '\n';
}

bool is_even(int n){ return n%2 == 0;}

int main(){

list <int> gqlist1, gqlist2;

for(int i = 0; i < 8; ++i){
  gqlist1.push_back(i * 2);
}
int mas[] = {1,2,5,3,4};
```

```
gqlist2.assign(mas,mas+5);

cout << "\nList 1 (gqlist1) is : ";
showlist(gqlist1);

cout << "\nList 2 (gqlist2) is : ";
showlist(gqlist2);

cout << "\ngqlist1.front() : "<< gqlist1.front();
cout << "\ngqlist1.back() : "<< gqlist1.back();

cout << "\ngqlist1.pop_front() : ";
gqlist1.pop_front();
showlist(gqlist1);

gqlist1.insert(gqlist2.begin(), 2, 10); // insert 2 tens
//gqlist1.insert(gqlist1.begin()+5, 100);  // iterator can't do
it
list <int>::iterator it = gqlist1.begin();
advance(it,5);
gqlist1.insert(it, 100);
showlist(gqlist1);

cout << "\ngqlist2.pop_back() : ";
gqlist2.pop_back();
showlist(gqlist2);

cout << "\ngqlist1.reverse() : ";
gqlist1.reverse();
showlist(gqlist1);

cout << "\ngqlist2.sort(): ";
gqlist2.sort();
showlist(gqlist2);

cout << "\ngqlist1.splice with list2: ";
list <int>::iterator ind = gqlist2.begin();
advance(ind,3);
gqlist1.splice(gqlist1.begin(),gqlist2, gqlist2.begin(), ind);
showlist(gqlist1);
cout << "\n list2after splice: ";
showlist(gqlist2);
```

```
cout << "\ngqlist1.unique: ";
gqlist1.unique(); // delete repeated values
showlist(gqlist1);

gqlist1.sort();
gqlist1.unique();
for(list<int>::iterator                    it                    =
gqlist1.begin();it!=gqlist1.end();++it){ (*it)++;}
cout<<"qlist1 update: ";
showlist(gqlist1);

gqlist1.merge(gqlist2);
cout<<"qlist1 & qlist2 merged: ";
showlist(gqlist1);

gqlist1.remove_if(is_even);
cout<<"removed even elements";
showlist(gqlist1);
}
```

Результат:

```
List 1 (gqlist1) is : 0,2,4,6,8,10,12,14
List 2 (gqlist2) is : 1,2,5,3,4
gqlist1.front() : 0
gqlist1.back() : 14
gqlist1.pop_front() : 2,4,6,8,10,12,14
2,4,6,8,10,100,12,14
gqlist2.pop_back() : 10,10,1,2,5,3
gqlist1.reverse() : 14,12,100,10,8,6,4,2
gqlist2.sort(): 1,2,3,5,10,10
gqlist1.splice with list2: 1,2,3,14,12,100,10,8,6,4,2
 list2after splice: 5,10,10
gqlist1.unique: 1,2,3,14,12,100,10,8,6,4,2
qlist1 update: 2,3,4,5,7,9,11,13,15,101
qlist1 & qlist2 merged: 2,3,4,5,5,7,9,10,10,11,13,15,101
removed even elements3,5,5,7,9,11,13,15,101
```

9. Стандартна бібліотека шаблонів STL. Ітератори та асоціативні контейнери

Вступ до ітераторів у C++

Ітератор – це об'єкт (як вказівник), який вказує на елемент всередині контейнера. Ми можемо використовувати ітератори для переміщення по вмісту контейнера. Їх можна уявити як щось подібне до вказівника, що вказує на певне місце, і ми можемо отримати доступ до вмісту з цього конкретного місця.
Ітератори забезпечують доступ до елементів контейнера. За допомогою ітераторів дуже зручно перебирати елементи. Ітератор описується типом iterator. Але для кожного контейнера конкретний тип ітератору буде відрізнятися. Так, ітератор для контейнеру list <int> представляє собою тип list <int> :: iterator, а ітератор контейнеру vector <int> представляє собою тип vector <int> :: iterator і так далі. Для отримання ітераторів контейнери в C++ мають такі методи, як begin() і end(). Функція begin () повертає ітератор, який вказує на перший елемент контейнера (при наявності в контейнері елементів). Функція end() повертає ітератор, який вказує на наступну позицію після останнього елемента, тобто по суті на кінець контейнера. Якщо контейнер порожній, то ітератори, які повертаються обома методами begin() і end() збігаються. Якщо ітератор begin не дорівнює ітератору end, то між ними є як мінімум один елемент. Обидві ці функції повертають ітератор для конкретного типу контейнеру.
Приклад:

```
int ar[]= { 1,2,3,4 }
std::vector<int> v(ar,ar+4) ;
std::vector<int>::iterator iter = v.begin();   // отримуємо ітератор
```

В даному випадку створюється вектор - контейнер типу vector, який містить значення типу int. І цей контейнер ініціалізується числами {1, 2, 3, 4}. І через метод begin() можна отримати ітератор для цього контейнера. Причому цей ітератор буде вказувати на перший елемент контейнеру.

Операції з ітераторами та види ітераторів

З ітераторами можна проводити наступні операції:

- *iter (розіменування) – отримання елементу, на який вказує ітератор. Якщо цей елемент має члени або методи то за допомогою оператору -> (або через оператори дужок, зірочка та крапка) можна отримати доступ до них безпосередньо з ітератору.
- ++iter(інкремент) – переміщення ітератору вперед для звернення до наступного елементу. Можлива як префіксна (рекомендовано) так і постфіксна форма).
- --iter (декремент) – переміщення ітератору назад для звернення до

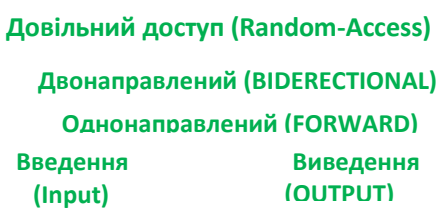
попереднього елемента. Можлива як префіксна (рекомендовано) так і постфіксна форма). Ітератори контейнера forward_list не підтримують операцію декременту.

- iter1 == iter2 (порівняння) – два ітератори рівні, якщо вони вказують на один і той самий елемент.
- iter1! = iter2 (негативне порівняння) – два ітератори не рівні, якщо вони вказують на різні елементи.
- Оператор присвоєння (оператор =) – присвоює ітератор (позицію елемента, на яку він посилається).
- Деякі ітератори підтримують додавання/віднімання цілого числа, порівняння(>, <) та оператор доступу квадратні дужки [].

Наприклад, використовуємо ітератори для перебору елементів вектору:

```
vector<int>::iterator iter = v.begin();   // отримуємо ітератор
while(iter!=v.end())   // поки не досягнемо кінця вектору
    {
        std::cout << *iter << std::endl; // виводимо результат як значення вказівника
        ++iter;  //  рухаємося  по  вектору  інкрементуючи ітератор
    }
```

Ітератори відіграють важливу роль у підключенні алгоритму з контейнерами разом з маніпуляціями даними, що зберігаються всередині контейнерів. Найбільш очевидною формою ітератору є вказівник. Вказівник може вказувати на елементи в масиві і може перебирати їх за допомогою оператору інкременту (++). Але не всі ітератори мають всю функціональність вказівників. Залежно від функціональності ітераторів вони можуть бути класифіковані на п'ять категорій, як показано на діаграмі нижче, причому зовнішнє є найпотужнішим, а отже, внутрішнє є найменш потужним з точки зору функціональності.



Не кожен з цих ітераторів підтримується всіма контейнерами в STL, різні контейнери підтримують різні ітератори, наприклад, вектори підтримують

ітератори довільного доступу, в той час як списки підтримують двонаправлені ітератори. Весь список наведено в таблиці:

Таблиця 8.1

Типи ітераторів

Контейнер	Тип підтримуваних ітераторів
Вектор (vector)	Довільного доступу
Список (list)	Двонаправлений
Дек (deque)	Довільного доступу
Масив (array)	Довільного доступу
Однонаправлений список (forward_list)	Однонаправлений ітератор
Відображення (map)	Двонаправлений
Мультивідображення (multimap)	Двонаправлений
Множина (set)	Двонаправлений
Мультимножина (multiset)	Двонаправлений
Стек (Stack)	Немає ітераторів
Черга (Queue)	Немає ітераторів
Черга з пріорітетами (Priority Queue)	Немає ітераторів

Таким чином, грунтуючись на функціональності ітераторів, вони можуть бути розділені на п'ять основних категорій:

1. Ітератори вводу (Input Iterators): Вони є найслабкішими з усіх ітераторів і мають дуже обмежену функціональність. Вони можуть використовуватися тільки в алгоритмах з одним проходом, тобто в тих алгоритмах, які обробляють контейнер послідовно, так що жоден елемент не доступний більш ніж один раз.
2. Ітератори виводу(**Output Iterators**): Так само, як і ітератори вводу, вони також дуже обмежені у своїй функціональності і можуть бути використані тільки в алгоритмі з одним проходом, але не для доступу до елементів, а для визначення(зміни) елементів.
3. Однонаправлений ітератор (**Forward Iterator**): Вони мають вищу ієрархію, ніж вхідні і вихідні ітератори, і містять всі функції, присутні в цих двох ітераторах. Але, як випливає з назви, вони також можуть рухатися лише у прямому напрямку, і це теж один крок за один раз.
4. Двонаправлені ітератори (**Bidirectional Iterators**): Вони мають всі особливості форвард ітераторів разом з але вони можуть рухатися в обох напрямках, тому їх звуть двонаправленими.
5. Ітератори прямого доступу (**Access Iterators**): Вони є найпотужнішими ітераторами. Вони не обмежуються переміщенням послідовно, як свідчить їх назва, вони можуть безпосередньо звертатися до будь-якого елемента

всередині контейнера. Їх функціональні можливості такі ж, як і у вказівників.

Наступна таблиця показує різницю в їх функціональності щодо різних операцій, які вони можуть виконувати.

Таблиця 8.2

Функціонали ітераторів

Тип ітератору	Доступ	Читання	Запис	Рух	Порівняння
Введення	->	= *i		++	==, !=
Виведення			*i=	++	
Форвард	->	= *i	*i=	++	==, !=
Двонаправлений		= *i	*i=	++, --	==, !=
Прямого доступу	->, []	= *i	*i=	++,--, +=, -=, +, -	==, !=, <, >,>=, <=

Можна побачити, що оператори які використовуються для ітераторів мають інтерфейс, що точно збігається з інтерфейсом звичайних вказівників у мовах С та С++, за допомогою яких можна обійти елементи і в звичайному масиві.
Різниця полягає в тому, що ітератор є інтелектуальним вказівником, тобто може обходити більш складні структури даних. Внутрішня поведінка ітераторів залежить від структури даних, по якій вони переміщаються. З цієї причини кожен контейнерний тип передбачає свій власний вид ітераторів. У результаті ітератори мають загальний інтерфейс, але різні типи. Це безпосередньо приводить до концепції узагальненого програмування: *операції використовують однаковий інтерфейс, але мають різні типи*, тому можна використовувати шаблони для формулювання узагальнених операцій, що застосовуються до довільних типів, що задовольняють зазначеному інтерфейсу.

Методи контейнерів для роботи з ітераторами

Усі контейнерні класи (послідовні контейнери та асоціативні контейнери) мають однакові основні функції-члени, що дозволяють переміщати ітератори по елементах контейнера.

Методи роботи з ітераторами:

- begin() – повертає ітератор на перший об’єкт вектору
- end() – повертає ітератор на останній об’єкт вектору
  Починаючі зі стандарту С++11 додано ще наступні варіанти доступу до ітераторів:
- rbegin() – повертає ітератор на останній об’єкт вектору як на початковий (reverse beginning). Рухається з останнього елементу до першого;
- rend() – повертає ітератор на перший об’єкт вектору як на останній (reverse beginning). Рухається з останнього елементу до першого;

- cbegin() – повертає константний ітератор на перший елемент;
- cend() – повертає константний ітератор на останній елемент;
- crbegin() – повертає константний реверсивний оператор на початок;
- crend() – повертає константний реверсивний оператор на кінець вектору.

Функція begin()повертає ітератор, що представляє початок контейнера, тобто позицію першого елемента, якщо такий мається в контейнері.
Функція end() повертає ітератор, що представляє кінець контейнера, тобто позицію, що слідує за останнім елементом.
Такий ітератор називається позамежним (past-the-end iterator).

Таким чином, функції – члени begin() і end() визначають напіввідкритий діапазон (half-open range), що включає перший елемент і не включає останній.
Напіввідкритий діапазон має дві переваги.
1. Існує простий критерій зупинки циклу при обході всіх елементів: цикл продовжується, поки не буде досягнута позиція end().
2. Він дозволяє уникнути спеціальної обробки порожніх діапазонів. Для порожніх діапазонів позиція begin() збігається з позицією end(). Наступний приклад демонструє використання ітераторів для виводу на екран всіх елементів списку(це варіант попереднього прикладу, але з використанням ітераторів).
Приклад:

```
#include<iostream>
#include<iterator> // Використовуємо iterator, begin() and end()
для ітераторів
#include<vector> //

int main() {

  int mas[8] ={1,2,3,4,5, 10,20,30};
  std::vector<int> ar(mas,mas+8);

  // Декларуємо ітератор до вектору
  std::vector<int>::iterator ptr;

  // виведемо елементи з допомогою begin() and end()
  std::cout << "vector: ";
  for(ptr = ar.begin(); ptr < ar.end(); ptr++)
    std::cout << *ptr << " ";
}
```

З C++11 для отримання константного ітератору const_iterator можна використовувати та методи отримання cbegin() та cend():

```
    std::vector<int> v { 1, 2, 3, 4, 5 }; // C++11 конструктор
       //std::vector<int>::const_iterator   iter;   //   тип
ітератору можна не визначати
    for (auto iter = v.cbegin(); iter != v.cend(); ++iter){
        std::cout << *iter << " ";
        // це неможливо бо ітератор константний
        //*iter = (*iter) * (*iter);
    }
    std::cout << std::endl;
```
Результат:
```
1 2 3 4 5
```

Також зауважимо, що з C++11 зявилася можливість ініціалізувати вектор за допомогою прямого присвоювання:
```
std::vector<int> v = { 1, 2, 3, 4, 5 }; // C++11
```

Реверсивні ітератори (C++11)

Реверсивні ітератори дозволяють перебирати елементи контейнеру в зворотному напрямку. Для отримання реверсивного ітератору використовують методи контейнерів rbegin() та rend(), а сам ітератор утворює тип reverse_iterator.
Так саме для реверсивного ітератору існує константна форма const_reverse_iterator яку можна отримати за допомогою методів crbegin() и crend():

```
std::vector<int> v { 1, 2, 3, 4, 5 };
        for   (std::vector<int>::reverse_iterator   iter   =
v.rbegin(); iter != v.rend(); ++iter){
         std::cout << *iter << " ";
              *iter = (*iter) * (*iter);
    }
    std::cout << "\n";

      for  (std::vector<int>::const_reverse_iterator  iter  =
v.crbegin(); iter != v.crend(); ++iter)
    {
        std::cout << *iter << " ";
        // неможливо бо ітератор константний
        //*iter = (*iter) * (*iter);
    }
```

Результат:

```
5 4 3 2 1
25 16 9 4 1
```

Функції роботи з ітераторами бібліотеки iterator

В бібліотеці <iterator> визначені деякі додаткові до стандартних функцій корисні функції для маніпулювання з ітераторами.
Серед них можна виділити наступні:

- advance(iter, n) - інкрементує ітератор iter на визначену кількість позицій n.

```
int mas[] ={1,2,3,4,5, 10,20,30};

std::vector<int> ar(mas,mas+5);
std::vector<int>::iterator ptr = ar.begin();
std::advance(ptr, 3);
//ptr += 3; // буде виконувати те саме
std::cout << "Third element is : "<<*ptr;

std::list<int> list1(mas,mas+5);
std::list<int> list2(mas+6,mas+8);
std::list<int>::iterator ptr2 = list1.begin();
std::advance(ptr2, 3);
//ptr += 3; // А ось для списку цей варіант некоректний
std::cout << "Third element is : "<<*ptr2;
```

- inserter(cont, iter) - функція для вставки елементів в будь-яку позицію контейнеру. Аргументи — контейнер та ітератор на позицію куди треба вставляти елементи. Функція повертає insert_iterator який дозволяє вставити елементи в інший контейнер.

```
#include<vector> // for vectors
#include<deque> // for deque
#include<list> // for list
// друк елементів будь-якої колекції
template<class Coll> void printCollect(Coll & v){
  typename Coll::const_iterator it = v.begin();
  for(;it!=v.end();++it){
    std::cout<<*it<<" ";
 }
 std::cout<<"\n";
}
```

```
int main() {
int mas[] ={1,2,3,4,5, 10,20,30};
std::vector<int> resulting(mas,mas+5); // куди вставляємо
std::vector<int> to_insert(mas+6,mas+8); // що вставляємо
std::vector<int>::iterator ptr = resulting.begin();
std::advance(ptr, 3); // на 3 позицію встановлюємо ітератор
//ptr += 3; // для вектору це те саме що і попередній рядок
std::cout << "Third element is : "<<*ptr<<"\n";
std::insert_iterator<std::vector<int>      >      ari      =
inserter(resulting, ptr); // куди вставляти
//std::insert_iterator<std::vector<int> > ari(resulting, ptr);
// копіюємо вміст to_insert(без 1-го ел-ту) в resulting
std::copy(to_insert.begin()+1, to_insert.end(), ari);
std::cout << "The new vector after inserting elements is : ";
printCollect(resulting);
// те саме для списку
std::list<int> list1(mas,mas+5);
std::list<int>::iterator ptr2 = list1.begin();
std::advance(ptr2, 3);
//ptr += 3; // а ось для списку це некоректно
std::cout << "Third element is : "<<*ptr2<<"\n";
std::copy(to_insert         .begin(),         to_insert.end(),
inserter(list1,ptr2));
std::cout << "The new list after inserting elements is : ";
printCollect(list1);
}
```

Результат:

```
Third element is : 4
The new vector after inserting elements is : 1 2 3 30 4 5
Third element is : 4
The new list after inserting elements is : 1 2 3 20 30 4 5
```

- back_inserter(), front_inserter(). Аналогічно до функції inserter визначені функції back_inserter(), front_inserter() які вставляють відповідно в початок та кінець колекції.

```
 std::deque<int> foo,bar;
  for    (int    i=1;    i<=5;    i++){    foo.push_back(i);
bar.push_back(i*10); }

  std::copy (bar.begin(),bar.end(),std::front_inserter(foo));
```

```
  std::cout << "foo after front insert contains:";
  printCollect(foo);

  std::vector<int> foo1,bar1;
  for (int i=1; i<=5; i++)  {
  foo1.push_back(i); bar1.push_back(i*10);
 }
  std::copy (bar1.begin(),bar1.end(),back_inserter(foo1));

  std::cout << "foo after back insert contains:";
  printCollect(foo1);
```

Результат:

```
foo after front insert contains:50 40 30 20 10 1 2 3 4 5
foo after back insert contains:1 2 3 4 5 10 20 30 40 50
```

- distance

Також можна вказати функцію distance, що визначає відстань між двома ітераторами:

```
std::vector<int> v(mas+1,mas+7);
    std::cout << "distance(first, last) = "
              << std::distance(v.begin(), v.end()) << '\n'
              << "distance(last, first) = "
              << std::distance(v.end(), v.begin()) << '\n';
               //поведінка цього виклику була невизначена до
C++11
Результат:
distance(first, last) = 6
distance(last, first) = -6
```

З C++11 додано також наступні функції:

- begin(coll) — повертає ітератор на початок колекції;
- end(coll) - повертає ітератор на кінець колекції;
- next(iter, n) - функція повертає новий ітератор, що вказує на позицію яка слідує за тією що стала після застосування n разів інкременту для iter;
- prev(iter, n) - функція повертає новий ітератор, що вказує на позицію яка передує той позиції, що стала після застосування n разів декременту для iter.

```
#include<iostream>
```

```
#include<iterator> // для методів iterator(advance)
#include<vector> // для vector

template <class Coll>
void printColl(const Coll & v){
 for(const auto & x: v){
  std::cout<< x <<" ";
}
std::cout<<"\n";
}

int main(){
 std::vector<int> ar = { 1, 2, 3, 4, 5 };

 //декларація початкового та кінцевого ітератору
 std::vector<int>::iterator ptr = begin(ar);
 std::vector<int>::iterator ftr = end(ar);

  //  next() для отримання нового ітератору
  // що вказує на 4
  auto it = next(ptr, 3);

  // prev() для отримання нового ітератору
  // що вказує на 3
  auto it1 = prev(ftr, 3);

  // позиція ітертору
  std::cout << "New iterator using next() position: ";
  std::cout << *it << " \n";

  // Displaying iterator position
  std::cout << "New iterator using prev() position: ";
  std::cout << *it1 << " \n";


  int arrayInt[] = {10,20,30,40,50, 60};
  std::vector<int> vectorInt;

  // iterate arrayInt: dcfdrf vectorInt
 // Обережно: вимагається об розмір arrayInt був парним!!!!!
  for (auto it = std::begin(arrayInt); it!=std::end(arrayInt);
std::advance(it,2))
```

```
    vectorInt.push_back(*it);

  // iterate bar: print contents:
  std::cout << "vector contains:";

  for(auto &x:vectorInt)std::cout << x <<" ";
}
```
Результат:
```
New iterator using next() position: 4
New iterator using prev() position: 3
vector contains:10 30 50
```

Створення власного ітератору

Оскільки ми знаймо методи ітераторів, то неважко власноруч створити власний ітератор.

```
#include <iostream>

#define MAXSIZE 100
// власний шаблон-огортка над масивом
template<class Type>class MyMasiv{
  Type mas[MAXSIZE];
  int size;
  public:
// створимо власний ітератор для цього класу
     class iterator{
          Type *current;
          public:
           // перевантажимо оператори
            iterator() { current = 0; }
            void operator++() {
                   current++;
            }
            void operator+=(int temp) {
                   current += temp;
            }
            void operator-=(int temp) {
                   current -= temp;
            }
            void operator=(Type& temp) {
```

```
                current = &temp;
            }
            bool operator!=(Type& temp) {
                return current != &temp;
            }
            bool operator==(Type& temp) {
                return current == &temp;
            }
           // будемо виводити всі числа поділивши їх на 2
            Type operator *() {
               return *current / 2;
            }
           Type* operator ->() {
              return current; }
           };
    // методи нашого масиву
      MyMasiv(){
           size = 0;
       }

     void add(int temp){
          size++;
         mas[size - 1] = temp;
     }
    void del(){
        size--;
       mas[size + 1] = 0;
    }
    void show(){
       std::cout << "Array:\n";
       for (int i = 0;i < size;i++)
            std::cout << mas[i] << ' ';
     std::cout << "\n";
     }
   // методи для отримання ітераторів
    Type& begin() { return mas[0]; }
    Type& end() { return mas[size]; }
};


int main(){
 MyMasiv<int> a;
```

```
 for (int i = 0;i < 5;i++){
     a.add(i*2);
  }

 MyMasiv<int>::iterator it;
 for (it = a.begin();it!=a.end();++it){
    std::cout << *it << ' ';
 }
}
```

Результат:

```
0 1 2 3 4
```

Асоціативні контейнери

Асоціативні контейнери — це контейнери які зберігають відсортовані дані використовуючи їх хеш-значення. Такі контейнери містять Сі++ реалізацію зокрема таких колекцій як Hashtable(Java) та Dictionary(Python), тобто це структури які оптимізовані під пошук у великій кількість даних та там де неважливий порядок введення даних, але важливе їх значення та порядок у якому вони будуть зберігатись.

Множина (Set)

Множина (Set) - це тип асоціативних контейнерів, в яких кожен елемент повинен бути унікальним, оскільки значення елемента його ідентифікує. Значення елемента не може бути змінено після його додавання до набору, хоча можна видалити та додати змінене значення цього елемента. Методи, які пов’язані з Set можуть бути розбити на наступні групи:

- **Ідентифікатори та модифікатори обсягу** ()
- Модифікатори даних
- Методи пошуку
- Робота з ітераторами

До С++11 шаблон класу множина мав наступні конструктори
1) конструктор за замовченням (default constructor), що створює порожню множину
2) Копіконструктор — копіює вміст іншого деку
3) Конструктор за інтервалом (range constructor) — конструює множину за двома ітераторами іншої колекції.

Ідентифікатори та модифікатори обсягу (Capacity)

- size() – кількість елементів колекції;
- max_size() – максимально можлива кількість елементів колекції;
- empty() – якщо порожня колекція повертає true, інакше false.

Модифікатори даних (Modification):

- insert(const g) – додає новий елемент ‘g’ до множини;
- insert (iterator position, const g) – додає елемент ‘g’ в позицію вказану ітертором;
- erase(iterator position) – видаляє елемент з позиції вказаної ітератором;
- erase(const g)– видаляє значення ‘g’ з множини;
- clear() – видаляє всі елементи колекції.

Починаючи з C++11 додали також наступні методи:

- emplace()– вставляє елемент в множину, якщо цього елементу ще в неї міститься;
- emplace_hint()– повертає ітератор на елемент, який вставляється в множину, якщо цього елементу до цього там не було;
- swap()– обмінює вміст двох множин ода в іншу.

Методи пошуку (Searching) та спостереження (Observation):

- key_comp() / value_comp() – повертає об’єкт, що вказує як елементи множини впорядкована (‘<‘ за замовченням).
- find(const g) – повертає ітератор на елемент ‘g’ якщо він є у множині, або ітератр на кінець множини, якщо його немає.
- count(const g) – повертає 1 або 0 в залежності від того є ‘g’ в множині або немає.
- lower_bound(const g) – повертає ітератор на перший елемент, що еквіалентний ‘g’ або на перший елемент, що точно більший за ‘g’;
- upper_bound(const g) – повертає ітератор на перший елемент, що еквіалентний ‘g’ або на останній елемент, що точно менший за ‘g’;
- equal_range((const g) – повертає ітератор на тип пара (key_comp) в якому перше поле pair::first яквівалентне lower bound, а pair::second еквівалентне upper bound.

Робота з ітераторами (iterators) — аналогічно до послідовних контейнерів:

- begin() – повертає ітератор на перший об’єкт вектору
- end() – повертає ітератор на останній об’єкт вектору

Починаючі зі стандарту C++11 додано ще наступні варіанти доступу до ітераторів:

- rbegin() – повертає ітератор на останній об’єкт вектору як на початковий (reverse beginning). Рухається з останнього елементу до першого;
- rend() – повертає ітератор на перший об’єкт вектору як на останній (reverse beginning). Рухається з останнього елементу до першого;
- cbegin() – повертає константний ітератор на перший елемент;

- cend() – повертає константний ітератор на останній елемент;
- crbegin() – повертає константний реверсивний оператор на початок;
- crend() – повертає константний реверсивний оператор на кінець вектору.

Крім того, для множини (як і для інших колекцій) перевизначений оператор присвоєння (operator=).

```
#include <iostream>
#include <set>
#include <iterator>

using namespace std;
// function comarator – порядок визначається квадратом числа
bool fncomp (int lhs, int rhs) {return lhs*lhs<rhs*rhs;}

//class Comparator – порядок по спаданню, а не зростанню
struct Classcomp {
  bool operator() (const int& lhs, const int& rhs) const
  {return lhs>rhs;}
};
// для множини з одним параметром конструктора
template<typename T>
void printSet(const set<T> v){
   typename set<T>::iterator itr;
   for(itr=v.begin(); itr!=v.end();++itr){
     cout <<" "<<*itr;
   }
    cout<< endl;
}
// для множини з двома параметрами конструктору
template<typename T, class U>
void printSet(const set<T,U> v){
   typename set<T,U>::iterator itr;
   for(itr=v.begin(); itr!=v.end();++itr){
     cout <<" "<<*itr;
   }
    cout<< endl;
}

int main (){
```

```
   set<int> first_set;                              //
конструктор за замовченням
   int myints[]= {10,-20,30,40,50,50,40};
   set<int> second_set(myints,myints+7);        // range-
конструктор
   set<int> third_set (second_set);                  //
копіконструктор
// конструктор по ітераторам
   set<int> fourth_set (third_set.begin(), third_set.end());
   cout<<"2,3,4 set:";
   printSet(fourth_set);

   cout<<"set size:"<< second_set.size()<<endl;
   cout<<"set max size:"<< second_set.max_size()<<endl;
   cout<<"how many of 20 there"<< second_set.count(20)<< " of
" << *second_set.find(20)<<endl;
   cout<<"how many of 50 there"<< second_set.count(50)<<" of "
<< *second_set.find(50)<<endl;
   cout<<"equal range for 20: " <<
*second_set.equal_range(20).first << ", " <<
*second_set.equal_range(20).second<<endl;

   set<int,Classcomp> fifth_set;                  // множина
відсортована Compare

   bool(*fn_pt)(int,int) = fncomp;
// множина відсортована по вказівнику на функцію fncomp
   set<int,bool(*)(int,int)> sixth_set (fn_pt);
 set <int,greater <int> > seventh_set;
  //вставка елементів
  seventh_set.insert(40);
  seventh_set.insert(30);
seventh_set.insert(60);
seventh_set.insert(-20);
seventh_set.insert(50);
seventh_set.insert(30); // лише єдине 30 буде додано в множину
seventh_set.insert(10);
 // вставка у множини
  for (set<int,greater <int> >::iterator it =
seventh_set.begin(); it!=seventh_set.end();++it){
    first_set.insert(*it);
    fifth_set.insert(*it);
```

```
    sixth_set.insert(*it);
  }
  cout<<"1 and 7th set:";
  printSet(first_set);
  cout<<"5th set:";
  printSet(fifth_set);
  cout<<"6th set:";
  printSet(sixth_set);

// видалити всі елементи до 30 в second_set
  cout << "second_set after removal of elements less than
30:";
  second_set.erase(second_set.begin(), second_set.find(30));
  printSet(second_set);
   // видалити елемент 50 в  third_set
  int num;
  num = third_set.erase (50);
  cout << "third_set.erase(50) : ";
  cout << num << " removed " ;
  printSet(third_set);

//lower bound та upper bound для seventh_set
cout << "seventh_set.lower_bound(40) : "<<
*seventh_set.lower_bound(40) << endl;
cout << "seventh_set.upper_bound(40) : "<<
*seventh_set.upper_bound(40) << endl;

// lower bound та upper bound для second_set
cout << "second_set.lower_bound(40) : "<<
*second_set.lower_bound(40) << endl;
cout << "second_set.upper_bound(40) : "<<
*second_set.upper_bound(40) << endl;
}
```

Результат:

```
2,3,4 set:    -20  10   30   40   50
set size:5
set max size:461168601842738790
how many of 20 there 0 of 5
how many of 50 there 1 of 50
equal range for 20: 30, 30
1 and 7th set:-20  10   30   40   50   60
5th set: 60   50   40   30   10   -20
```

```
6th set: 10   -20  30   40   50   60
second_set after removal of elements less than 30:  30   40
     50
third_set.erase(50) : 1 removed  -20  10   30   40
seventh_set.lower_bound(40) : 40
seventh_set.upper_bound(40) : 30
second_set.lower_bound(40) : 40
second_set.upper_bound(40) : 50
```

Мультимножини (Multiset)

Мультимножини - це тип асоціативних контейнерів, який схожий на множини, але відмінність полягає в тому, що елементи можуть повторюватись (тобто не обов'язкова унікальність елементів).
Методи для роботи з мультимножиною класифікуються так само як і для множини:

- **Ідентифікатори та модифікатори обсягу** ()
- Модифікатори даних
- Методи пошуку
- Робота з ітераторами

До C++11 шаблон класу множина мав наступні конструктори
1) конструктор за замовченням (default constructor), що створює порожню множину
2) Копіконструктор — копіює вміст іншого деку
3) Конструктор за інтервалом (range constructor) — конструює множину за двома ітераторами іншої колекції.

Ідентифікатори та модифікатори обсягу (Capacity):

- size() – кількість елементів колекції;
- max_size() – максимально можлива кількість елементів колекції;
- empty() – якщо порожня колекція повертає true, інакше false.

Модифікатори даних (Modification):

- insert(const g) – додає новий елемент 'g' до мультимножини;
- insert (iterator position, const g) – додає елемент 'g' в позицію вказану ітертором;
- erase(iterator position) – видаляє елемент з позиції вказаної ітератором;
- erase(const g) – видаляє значення 'g' з мультимножини;
- clear() – видаляє всі елементи колекції.

Починаючи з C++11 додали також наступні методи:

- emplace() – вставляє елемент в мультимножину, якщо цього елементу ще в неї міститься;
- emplace_hint() – повертає ітертор на елемент, який вставляється в множину, якщо цього елементу до цього там не було;
- swap() – обмінює вміст двох множин одну в іншу.

Методи пошуку (Searching):

- key_comp() / value_comp() – повертає об'єкт, що вказує як елементи множини впорядкована ('<' за замовченням);
- find(const g) – повертає ітератор на елемент 'g' якщо він є у множині, або ітератр на кінець множини, якщо його немає;
- count(const g) – повертає кількість входжень елементу 'g' в мультимножину;
- lower_bound(const g) – повертає ітератор на перший елемент, що еквіалентний 'g' або на перший елемент, що точно більший за 'g';
- upper_bound(const g) – повертає ітератор на перший елемент, що еквівалентний 'g' або на останній елемент, що точно менший за 'g';
- equal_range((const g) – повертає ітератор на тип пара (key_comp) в якому перше поле pair::first еквівалентне lower bound, а pair::second еквівалентне upper bound.

Робота з ітераторами (iterators) — аналогічно до множини та послідовних контейнерів.

Крім того, для множини (як і для інших колекцій) перевизначений оператор присвоєння (operator=).

```
#include <iostream>
#include <set>
#include <vector>

// function comarator
bool fncomp (int lhs, int rhs) {return lhs*lhs<rhs*rhs;}

//class Comparator
struct Classcomp {
  bool operator() (const int& lhs, const int& rhs) const
  {return lhs>rhs;}
};

template<class T>
void printSet(const T & v){
   typename T::iterator itr;
   for(itr=v.begin(); itr!=v.end();++itr){
```

```
     std::cout <<" "<<*itr;
   }
    std::cout<< "\n";
}

int main (){
   std::multiset<int> first_set;                              //
empty set of ints
   int myints[]= {10,-20,30,40,50,50,40};

   std::multiset<int> second_set(myints,myints+7);            //
range
   std::multiset<int> third_set (second_set);                 //
a copy of second
   //set<int> fourth_set (third_set.begin(), third_set.end());
// iterator constructor
   std::vector<int> v(third_set.begin(), third_set.end());
   std::multiset<int> fourth_set (v.begin(), v.end());
   std::cout<<"2,3,4 set:";
   printSet(fourth_set);

   std::cout<<"set size:"<< second_set.size()<<"\n";
   std::cout<<"set max size:"<< second_set.max_size()<<"\n";
   std::cout<<"how many of 20 there "<< second_set.count(20)<<
" of " << *second_set.find(20)<<"\n";
   std::cout<<"how many of 50 there "<< second_set.count(50)<<"
of " << *second_set.find(50)<<"\n";
   std::cout<<"equal      range      for      20:      "      <<
*second_set.equal_range(20).first      <<      ",      "      <<
*second_set.equal_range(20).second<<"\n";

   std::multiset<int,Classcomp> fifth_set;                    //
class as Compare

   bool(*fn_pt)(int,int) = fncomp;
   std::multiset<int,bool(*)(int,int)> sixth_set (fn_pt);   //
function pointer as Compare

   std::multiset <int,std::greater<int> > seventh_set;
  //insert elements in random order
  seventh_set.insert(40);
  seventh_set.insert(30);
```

```
  seventh_set.insert(60);
  seventh_set.insert(-20);
  seventh_set.insert(50);
  seventh_set.insert(30); // only one 30 will be added to the
set
  seventh_set.insert(10);
  for (std::multiset<int,std::greater <int> >::iterator it =
seventh_set.begin(); it!=seventh_set.end();++it){
    first_set.insert(*it);
    fifth_set.insert(*it);
    sixth_set.insert(*it);
  }
  std::cout<<"1 and 7th set:";
  printSet(first_set);
  std::cout<<"5th set:";
  printSet(fifth_set);
  std::cout<<"6th set:";
  printSet(sixth_set);

// remove all elements up to 30 in
  std::cout << "second_set after removal of elements less than
30:";
  second_set.erase(second_set.begin(), second_set.find(30));
  printSet(second_set);
   // remove element with value 50 in
  int num;
  num = third_set.erase (50);
  std::cout << "third_set.erase(50) : ";
  std::cout << num << " removed " ;

  printSet(third_set);

  //lower bound and upper bound for set seventh_set
  std::cout    <<    "seventh_set.lower_bound(40)    :    "<<
*seventh_set.lower_bound(40) << "\n";
  std::cout    <<    "seventh_set.upper_bound(40)    :    "<<
*seventh_set.upper_bound(40) << "\n";

  //lower bound and upper bound for set second_set
  std::cout    <<    "second_set.lower_bound(40)    :    "<<
*second_set.lower_bound(40) << "\n";
```

```
  std::cout    <<    "second_set.upper_bound(40)    :    "<<
*second_set.upper_bound(40) << "\n";
}
```
Результат:
```
2,3,4 set:    -20  10   30   40   40   50   50
set size:7
set max size:461168601842738790
how many of 20 there 0 of 7
how many of 50 there 2 of 50
equal range for 20: 30, 30
1 and 7th set:-20  10   30   30   40   50   60
5th set:  60   50   40   30   30   10   -20
6th set:  10   -20  30   30   40   50   60
second_set after removal of elements less than 30:  30   40
     40   50   50
third_set.erase(50) : 2 removed  -20  10   30   40   40
seventh_set.lower_bound(40) : 40
seventh_set.upper_bound(40) : 30
second_set.lower_bound(40) : 40
second_set.upper_bound(40) : 50
```

Зауважимо, що другий необов’язковий параметр у конструкторі множини або мультимножини — це так званий функціональний об’єкт, що вказує на те за яким критерієм сортуються значення в множині.
Так для впорядкування множини у спадаючому порядку можна записати:
```
typedef set<int,greater<int>> IntSet;
```

Об'єкт greater<>— це стандартний функціональний об'єкт, що вказує, що об’єкти сортуються за зростанням (докладніше про це — в розділі про функтори).

Відображення (Map)

Відображення — асоціативний контейнер, який зберігає дані у вигляді пари (ключ, значення). При цьому дані відсортовані по значенню ключа та кожен ключ — унікальний.

Методи для роботи з відображенням класифікуються так само як і для множини:

- Ідентифікатори та модифікатори обсягу
- Модифікатори даних
- Методи пошуку
- Робота з ітераторами

До С++11 шаблон класу відображення мав наступні конструктори
1) конструктор за замовченням (default constructor), що створює порожню множину
2) Копіконструктор — копіює вміст іншого деку
3) Конструктор за інтервалом (range constructor) — конструює відображення за двома ітераторами на колекцію, що складається з пар об'єктів.

Ідентифікатори та модифікатори обсягу (Capacity)

- size() – кількість елементів колекції;
- max_size() – максимально можлива кількість елементів колекції;
- empty() – якщо порожня колекція повертає true, інакше false.

Модифікатори даних (Modification):

- insert(const pair g) – додає нову пару ключ/значення 'g' до відображення;
- оператор "квадратні дужки" (operator[key])– отримує доступ до значення за ключем;
- at(key) - отримує доступ до значення за ключем (з виключенням при некоректному доступі);
- erase(iterator position1, iterator position2) – видаляє елемент з інтервалу вказаного ітераторами;
- erase(const g)– видаляє значення 'g' з відображення;
- clear() – видаляє всі елементи колекції.

Починаючи з С++11 додали також наступні методи:

- emplace() – вставляє пару в множину, якщо цього елементу ще в неї міститься;
- emplace_hint() – повертає ітератор на елемент, який вставляється в відображення, якщо цього елементу до цього там не було;
- swap() – обмінює вміст двох відображень одна в іншу.

Методи пошуку (Searching):

- key_comp() / value_comp() – повертає об'єкт, що вказує як ключ/значення впорядковані ('<' за замовченням);
- find(const g) – повертає ітератор на значення ключа 'g' якщо він є у множині, або ітератор на кінець відображення, якщо його немає;
- count(const g) – повертає кількість входжень ключу 'g' в колекції;
- lower_bound(const g) – повертає ітератор на перший ключ, що еквівалентний 'g' або на перший елемент, що точно більший за 'g';
- upper_bound(const g) – повертає ітератор на перший ключ, що еквівалентний 'g' або на останній елемент, що точно менший за 'g';
- equal_range((const g) – повертає ітератор на тип пара (key_comp) в якому перше поле pair::first еквівалентне lower bound, а pair::second еквівалентне upper bound.

Робота з ітераторами (iterators): така сама як і для множини та контейнерів послідовного доступу.

```
#include <iostream>
#include <map>
#include <stdexcept>

using namespace std;

template <typename K, typename V>
void printMap(const map<K,V> & dict, string name){
    // друк відображення
    typename map<K, V>::const_iterator itr;
    cout << "The map " << name <<" is : \n";
    for (itr = dict.begin(); itr != dict.end(); ++itr) {
        cout << " " << itr->first
             << " " << itr->second << " ";
    }
    cout << endl;
}

int main(){
    // порожнє відображення
    map<int, int> dict1;

    // вставка довільних елементів
    dict1.insert(pair<int, int>(1, 40));
    dict1.insert(pair<int, int>(3, 30));
    dict1.insert(pair<int, int>(2, 60));
    dict1.insert(pair<int, int>(5, 20));
    dict1.insert(pair<int, int>(4, 50));
    dict1.insert(make_pair(7, 50));
    dict1.insert(make_pair(6, 10));
    //dict1.insert(make_pair(6,   15));   //   змінюємо   старе
значення

    printMap(dict1, "dict1");

    // конструктор по ітераторам
    map<int, int> dict2(dict1.begin(), dict1.end());

    // доступ та зміна елементів за значенням
```

```
    dict2[6] = 20;
    dict2.at(7) = 30;
    try{    // перевірка коректності доступу
     dict2.at(8) = 30;
    }
    catch(out_of_range & ex){
     cout<<"error Incorrect key"<<ex.what()<<endl;
    }

    printMap(dict2, "dict2");

    // видалимо всі елементи до key=3 з dict2
    cout << "dict2 after removal of elements less than key=3 :
";
    dict2.erase(dict2.begin(), dict2.find(3));
    printMap(dict2, "dict2 erased: ");

    // видалимо всі елементи з key = 5
    int num;
    num = dict2.erase(5);
    cout << "dict2.erase(5) : " << num << " removed \n";

    printMap(dict2, "dict2 erased 5");
    // lower bound та upper bound для map з key = 5
    cout << "dict1.lower_bound(5) : "  << "KEY = ";
    cout << dict1.lower_bound(5)->first << " ";
    cout << " ELEMENT  =  "<<  dict1.lower_bound(5)->second  <<
endl;
}
```

Результат:

```
The map dict1 is :
 1 40  2 60  3 30  4 50  5 20  7 50
error Incorrect index map::at
The map dict2 is :
 1 40  2 60  3 30  4 50  5 20  6 20  7 30
dict2 after removal of elements less than key=3 : The map dict2
erased:  is :
 3 30  4 50  5 20  6 20  7 30
dict2.erase(5) : 1 removed
The map dict2 erased 5 is :
 3 30  4 50  6 20  7 30
```

```
dict1.lower_bound(5) : KEY = 5   ELEMENT = 20
```

Мультивідображення (Multimap)

Мультивідображення відрізняється від відображення можливістю додавати пари значень з еквівалентними ключами. Це виключає можливість використання квадратних дужок для доступу до елементів, але розширює можливості додавання елементів.

```
#include <iostream>
#include <map>
#include <stdexcept>

using namespace std;

template <typename K, typename V>
void printMap(const multimap<K,V> & dict, string name){
    // друкуємо відображення
    typename multimap<K, V>::const_iterator itr;
    cout << "The map " << name <<" is : \n";
    for (itr = dict.begin(); itr != dict.end(); ++itr) {
        cout << " " << itr->first
             << " " << itr->second << " ";
    }
    cout << endl;
}

int main(){
    // конструктор за замовченням
    multimap<string, string> dict1;

    // вставляємо елементи
    dict1.insert(make_pair("1", "word1"));
    dict1.insert(make_pair("2", "word2"));
    dict1.insert(make_pair("1", "word3"));
    dict1.insert(make_pair("2", "word4"));
    dict1.insert(make_pair("3", "word5"));
    dict1.insert(make_pair("4", "word6"));
    dict1.insert(make_pair("5", "word7"));
    dict1.insert(make_pair("5", "word8"));

    printMap(dict1, "dict1");
```

```
    // конструктор за ітераторами
    multimap<string, string> dict2(dict1.begin(), dict1.end());

    printMap(dict2, "dict2");

    // видаляємо всі елементи з key=3 in dict2
    cout << "dict2 after removal of elements less than key=1 :
\n";
    dict2.erase(dict2.begin(), dict2.find("2"));
    printMap(dict2, "dict2 erased: ");

    // видаляємо елементи з key = 5
    int num;
    num = dict2.erase("4");
    cout << "dict2.erase(4) : " << num << " removed \n";

    printMap(dict2, "dict2 erased 4");
    // lower bound та upper bound для map з key = 5
    cout << "dict1.lower_bound(5) : "  << "KEY = ";
    cout << dict1.lower_bound("5")->first << " ";
    cout << " ELEMENT = "<< dict1.lower_bound("5")->second <<
endl;

    dict1.insert(make_pair("5", "word9"));
    dict1.insert(make_pair("6", "word10"));
    cout<<"all values with key=5:\n";
    pair                    <multimap<string,string>::iterator,
multimap<string,string>::iterator> eq_range;
    eq_range = dict1.equal_range("5");
    for  (multimap<string,string>::iterator  it=eq_range.first;
it!=eq_range.second; ++it)
      cout << " " << it->second;

}
```

Результат:

```
The map dict1 is :
 1 word1  1 word3  2 word2  2 word4  3 word5  4 word6  5 word7
5 word8
The map dict2 is :
 1 word1  1 word3  2 word2  2 word4  3 word5  4 word6  5 word7
5 word8
dict2 after removal of elements less than key=1 :
```

```
The map dict2 erased:  is :
 2 word2  2 word4  3 word5  4 word6  5 word7  5 word8
dict2.erase(4) : 1 removed
The map dict2 erased 4 is :
 2 word2  2 word4  3 word5  5 word7  5 word8
dict1.lower_bound(5) : KEY = 5   ELEMENT = word7
all values with key=5:
 word7 word8
```

Контейнери та методи, що були додані в C++11

Деякі корисні нововведення C++11

У відповідності зі стандартом C++11 ключове слово auto дозволяє вказати точний тип ітератору (за умови, що ітератор був ініціалізованим під час оголошення, так що його тип можна вивести з його початкового значення). Таким чином, безпосередня ініціалізація ітератору за допомогою функції begin() дозволяє використовувати ключове слово auto для оголошення його типу:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
     cout << *pos << ’ ’;
}
```

Легко бачити, що використання ключового слова auto робить код більш компактним.

Без ключового слова auto оголошення ітератору в циклі виглядало б у такий спосіб:

```
for (list<char>::const_iterator pos = coll.begin();
     pos != coll.end(); ++pos) {
     cout << *pos << ’ ’;
}
```

Інша перевага застосування auto полягає в тому, що цикл є стійким до змін коду, таким як модифікація типу контейнеру.

Однак у такої конструкції є недолік — ітератор втрачає свою константність, тобто з'являється ризик ненавмисного присвоювання. Вираз auto pos = coll.begin() робить ітератор pos неконстантним, тому що функція begin() повертає об'єкт типу контейнер::iterator.

Для того щоб зберегти константність ітератору, у стандарті C++11 передбачені функції

cbegin() і cend().

Вони повертають об'єкт типу контейнер::const_iterator.

Таким чином, в стандарті C++11 цикл, що дозволяє обходити всі елементи контейнеру без використання діапазонного циклу for, може мати наступний вигляд:

```
for (auto pos = coll.cbegin(); pos != coll.cend(); ++pos) {
...
}
```

Крім того, в Ci++11 з'явився цикл foreach типу, який дозволяє ітеруватись по контейнеру проходячи всі елементи колекції. Тобто, конструкція

```
for (type elem : coll) {
     ...
}
```

Інтерпретується як

```
for (auto pos=coll.begin(), end=coll.end(); pos!=end; ++pos) {
     type elem = *pos;
     ...
}
```

Приклади:

1) Виведення вектору можна записати наступним чином:

```
vector<string> v = {“aaa”,”bbb”, “cccc”};
for(string s: v){
  cout<<s;
}
```

Або більш універсально:

```
for(const auto & s: v){
  cout<<s;
}
```

Вставка контейнеру у контейнер

Всі асоціативні контейнери передбачають метод insert()для вставки нового елементу.

```
coll.insert(3);
coll.insert(1);
...
```

В C++11 можна просто:

```
coll.insert ( { 3, 1, 5, 4, 1, 6, 2 } );
```

Кожен вставлений елемент автоматично займає правильну позицію відповідно до критерію сортування.

Ініціалізація контейнеру

В C++11 додали більш зручну можливість ініціалізації контейнеру за допомогою фігурних дужок

```
vector<int> v{1,2,3,4,5};
```

Аналогічно, оскільки в нас є копіконструктор стола можлива і наступна ініціалізація

```
vector<int> v = {1,2,3,4,5};
```

Клас array

Введення класу масиву з C++ 11 запропонувало кращу альтернативу для масивів C-стилю. Переваги класу масиву над масивом C-стилю:
• Класи масивів знають свій власний розмір, тоді як масиви C-стилю не мають цієї властивості. Тому при переході до функцій нам не потрібно передавати розмір масиву як окремий параметр.
• З масивом стилів C більший ризик розпаду масиву в вказівник. Класи масивів не розпадаються на вказівники.
• Класи масивів, як правило, є більш ефективними, легкими та надійними, ніж масиви C-стилю.

Методи array :

- at() - доступ до елементу за його номером;
- get() - ця функція також дозволяє отримати доступ до елементу масиву, але це не метод контейнеру, а дружня функція класу tuple.
- operator[] - доступ до елементу.

```
// C++ код для демонстрації array:  to() та get()
#include<iostream>
#include<array> // для array, at()
#include<tuple> // для get()
using namespace std;
int main() {
// ініціалізація масиву
array<int,6> ar = {1, 2, 3, 4, 5, 6};

// виведення за допомогою at()
cout << "The array elemets are (using at()) : ";
for( inti=0; i<6; i++)
cout << ar.at(i) << " ";
cout << endl;
```

```
//виведення за допомогою  get()
cout << "The array elemets are (using get()) : ";
cout << get<0>(ar) << " "<< get<1>(ar) << " ";
cout << get<2>(ar) << " "<< get<3>(ar) << " ";
cout << get<4>(ar) << " "<< get<5>(ar) << " ";
cout << endl;

// виведення за допомогою operator[]
cout << "The array elements are (using operator[]) : ";
for( inti=0; i<6; i++)
cout << ar[i] << " ";
cout << endl;
```

- **front()** - повертає перший елемент масиву.
- **back()** - повертає останній елемент масиву

```
// C++  front() and back()
#include<iostream>
#include<array> // для front() та back()
using namespace std;
int main(){
// ініціалізація
array<int,6> ar = {1, 2, 3, 4, 5, 6};

// друк першого елементу
cout << "First element of array is : ";
cout << ar.front() << endl;

// друк останнього елементу
cout<< "Last element of array is : ";
cout << ar.back() << endl;
```

- **size()** - розмір масиву (цього методу не має в Сі-масивах).
- **max_size()** - максимально можливий розмір масиву.

```
// C++ код для size() та max_size()
#include<iostream>
#include<array> // size() та max_size()
Using namespace std;
Int main() {
array<int,6> ar = {1, 2, 3, 4, 5, 6};

// друк елементів
```

```
cout << "The number of array elements is : ";
cout << ar.size() << endl;

// максимальний розмір
cout << "Maximum elements array can hold is : ";
cout << ar.max_size() << endl;
```

- **empty()** - чи порожній масив;
- **fill()** - метод дозволяє заповнити масив певним значенням.

```
#include<iostream>
#include<array> // fill(), empty()
using namespace std;
int main() {
// декларація 1-го масиву
array<int,6> ar;

// декларація  2-го масиву
array<int,0> ar1;

// перевірка на порожнину
ar1.empty()? cout << "Array empty":
cout << "Array not empty";
cout << endl;

// заповнення масиву  нулями
ar.fill(0);

// виведення масиву
cout << "Array after filling operation is : ";
for( inti=0; i<6; i++)
cout << ar[i] << " ";
}
```

Клас forward_list

Однонаправлений список – в деяких ситуаціях працює швидше ніж двонаправлений список, але методи доступу та роботи з ітератором тут відрізняються, оскільки на відміну від інших контейнерів ітератор тут може рухатись лише в одному напрямку.

Методи контейнеру список

Модифікатори:

- operator= – оператор присвоєння;
- assign(n,val) – присвоює значення контейнеру;

Доступ до елементів:

- front() – доступ до першого елементу

Робота з ітераторами:

- before_begin() – повертає ітератор перед початком списку;
- cbefore_begin() – повертає константний ітератор перед початком списку;
- begin() - повертає початковий ітератор;
- cbegin() - повертає константний початковий ітератор;
- end() - повертає ітератор на кінець списку;
- cend() - повертає константний ітератор на кінець списку;

Ідентифікатори обсягу:

- empty() - чи порожній контейнер;
- max_size - максимальна кількість елементів контейнеру;

Модификатори:

- clear() – очищення вмісту контейнеру;
- insert_after() – вставка елементів після даного ітератору;
- emplace_after() – створення елементів після даного ітератору;
- erase_after() – видаляє елемент після ітератору;
- push_front() – вставляє елемент в початок;
- emplace_front() – створює елемент на початку;
- pop_front() – видаляє перший елемент;
- resize() – змінює кількість виділених елементів масиву;
- swap() – обмін змісту двох списків;
- merge() – об’єднує два відсортованих списки;
- splice_after () – переміщує елементи з іншого forward_list;
- remove(g) – видаляє даний елемент або список елементів;
- remove_if(func) – видаляє елементи по вказаному критерію (func - функція або функціональний об’єкт);
- reverse() – інвертує список;
- unique() – видаляє послідовні однакові елементи;
- sort() – сортує список.

```
#include <forward_list>
#include <string>
```

```
#include <iostream>

template<typename T>
std::ostream&    operator<<(std::ostream&    s,    const
std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main() {
    // c++11 initializer list syntax:
    std::forward_list<std::string>    words1    {"the",
"frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::forward_list<std::string>  words2(words1.begin(),
words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
    std::forward_list<std::string> words3(words1);
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::forward_list<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Результат:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

10. Функтори та алгоритми

Алгоритми

Однією з важливих властивостей та власно й цілей створення бібліотеки шаблонів було створення можливостей для стандартного використання різноманітних алгоритмів до різних типів даних. Деякі з цих алгоритмів вже містяться в деяких контейнерах. Зокрема, в контейнері List містяться методи сортування та злиття відсортованих списків, в асоціативних контейнерах є методи пошуку, а пріоритетна черга автоматично сортує дані. Однак, інколи було б бажано застосовувати сортування, бінарний пошук, знаходити загальну суму елементів колекції, застосовувати якусь функцію до кожного елементу контейнеру незалежно від типу контейнеру або взагалі застосувати її до звичайного масиву.

З цією метою до стандартної бібліотеки C++ було додано дві бібліотеки algorithms та numeric, які містять функції, що реалізують деякі з відомих та потрібних у практичному програмуванні алгоритмів.

Ці бібліотеки дозоляють не витрачати час на написання циклів для виконання популярних задач таких як сортування, пошук або підрахунок кількості елементів масиву. Це дозволяє також не витрачати час на оптимізацію та відлагодження цих алгоритмів. Крім того деякі з цих реалізацій дозволяють використовувати розпаралелювання – можливість виконувати їх в декілька потоків для пришвидшення.

Важливою особливістю стандартної бібліотеки C++ є те, що вона не тільки визначає синтаксис і семантику узагальнених алгоритмів, а й також має вимоги щодо їх продуктивності.

Функції бібліотеки algorithms

Функції бібліотеки algorithms поділяють на наступні категорії:

- немодифікуючі операції з послідовностями (non-modifying sequence operations);
- модифікуючі операції з послідовностями (modifying sequence operations);
- операції розділення (Партіції)(Partitions);
- сортування (Sorting);
- бінарний пошук (Binary search) на відсортованих або частково відсортованих послідовностях;
- мінімуми/максимуми (Min/max);
- злиття (Merge) на відсортованих послідовностях;
- операції на структурі Купа (Heap);
- інші.

Немодифікуючі операції з послідовностями

Немодифікуючі операції з послідовностями — це функції що виконують якусь певну дії над масивом даних, але не змінюють їх змісту. Зокрема, це методи пошуку, послідовного виконання дій тощо.
Можна виділити наступні підгрупи функцій в даному класі функцій:

Функція for_each

Функція for_each(pos1, pos2, fun) виконує функцію fun для всіх елементів, що знаходяться між двома ітераторами pos1 та pos2.
Приклад.

```
#include <iostream>     // std::cout
#include <algorithm>    // std::for_each
#include <vector>       // std::vector

template<typename T>
void myfunction (T i) {  // функція над елементом послідовності
  std::cout << " " << ((i>0)?"positive ":"negative ");
}

int main () {
  double mas[] = {1.0, 2.0, -4.0, 3.0};
  std::vector<int> myvector(mas,mas+4);

  std::cout << "myvector signs:";
 std::for_each        (myvector.begin(),        myvector.end(),
myfunction<int>);
  std::cout << "\narray signs:";
  std::for_each (mas, mas+4, myfunction<double>);

}
```

Результат:

```
myvector signs: positive  positive  negative  positive
array signs: positive  positive  negative  positive
```

Функція find

Функція find(pos1, pos2, g) шукає елемент g серед всіх елементів, що знаходяться між двома ітераторами pos1 та pos2. Якщо елемент знайдений — повертається ітератор вводу, що вказує на цей елемент, якщо ні — ітератор на кінцеву позицію. Для порівняння об’єктів функція використовує оператор рівності(==).
Приклад:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find
#include <vector>       // std::vector
```

```
int main () {

  int myints[] = { 10, 20, 30, -40, 30};
  int * p;
  // пошук в масиві – результат вказівник
  p = std::find (myints, myints+5, 30);
  if (p != myints+5)
    std::cout << "Element found in myints: " << *p << '\n';
  else
    std::cout << "Element not found in myints\n";

  double mas[] = {1.0, 2.0, -4.0, 3.0};
  std::vector<int> myvector(mas,mas+4);
  std::vector<int>::iterator it;
// пошук в векторі – результат ітератор
  it = std::find (myvector.begin(), myvector.end(), 30);
  if (it != myvector.end())
    std::cout << "Element found in myvector: " << *it << '\n';
  else
    std::cout << "Element not found in myvector\n";
}
```

Результат:

```
Element found in myints: 30
Element not found in myvector
```

Для функцій пошуку також можна використовувати форми find_if, find_end, find_first_of, adjacent_find, а з C++11 також додана функція find_if_not.

Приклад (adjacent_find — знаходження однакових сусідів):

```
// adjacent_find example
#include <iostream>     // std::cout
#include <algorithm>    // std::adjacent_find
#include <vector>       // std::vector

template<typename T>
bool myEqual (T x, T y) {
  return (x-y)*(x-y)<0.0001;
}

int main () {
  float                          myarr[]                          =
{5.0f,2.f,5.0f,3.0f,2.9999f,2.0f,1.0f,1.0001f,2.0f};
  std::vector<float> myvector2 (myarr,myarr+8);
```

```
  std::vector<float>::iterator it2;
  std::vector<int> myvector3(myarr,myarr+8);// 5,2,5,3,2,2,1,1
  std::vector<int>::iterator it3;

  // порівняння цілих за замовченням
  it3      =      std::adjacent_find      (myvector3.begin(),
myvector3.end());

  if (it3!=myvector3.end())
    std::cout << "the first value of repeated elements are: "
<< *it3 << '\n';

  //використовуючи власне порівняння-предикат
  it2 = std::adjacent_find (myvector2.begin(), myvector2.end(),
myEqual<float>);

  if (it2!=myvector2.end())
    std::cout << "the first value of almost repeated elements
are: " << *it2 << '\n';
}
```

Результат:

```
the first value of repeated elements are: 2
the first value of almost repeated elements are: 3
```

Примітка. В усіх випадках де використовується функція як аргумент — замість неї можна (а іноді й бажано) використовувати функтор (або лямбду в C++11).

Функція count

Функція count(pos1, pos2, g) повертає кількість входжень g серед всіх елементів, що знаходяться між двома ітераторами pos1 та pos2. Для порівняння об'єктів функція використовує оператор рівності(==).

Приклад:

```
// count algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::count
#include <vector>       // std::vector

int main () {
  // підрахуємо 10-ки серед елементів масиву:
  int myints[] = { 10, 20, 30, -40, 30, 30};
  double mas[] = {1.0, 2.0, -4.0, 3.0 };
  std::vector<int> myvector(mas,mas+4);
```

```
   int mycount = std::count (myints, myints+6, 30);
  std::cout << "10 appears " << mycount << " times.\n";

  // підрахуємо 20-ки серед елементів вектору:
  mycount = std::count (myvector.begin(), myvector.end(), 20);
  std::cout << "20 appears " << mycount  << " times.\n";;
}
```

Результат:

```
10 appears 3 times.
20 appears 0 times.
```

Також існує функція пошуку count_if, у формі count_if(pos1,pos2,fun) яка порівнює елементи за умовою, що задається предикатом fun.

Приклад:

```
// count_if
#include <iostream>     // std::cout
#include <algorithm>    // std::count_if
#include <vector>       // std::vector

bool isOdd (int i) {
  return ((i%2)==1);
}

int main () {
  std::vector<int> myvector4;
  for (int i=1; i<10; i++) myvector4.push_back(i); // myvector:
1 2 3 4 5 6 7 8 9

  int mycount = count_if (myvector4.begin(), myvector4.end(),
isOdd);
  std::cout << "myvector contains " << mycount   << " odd
values.\n";
}
```

Результат:

```
myvector contains 5 odd values.
```

Функція mismatch

Функція mismatch у формах mismatch(pos1,pos2, col) та mismatch(pos1,pos2, col,fun) знаходить елементи в двох послідовностях (контейнерах), що не співпадають. Перший контейнер задається ітераторами початкової та кінцевої позиції, а другий — своєю початковою позицією col. Опціональний параметр fun

вказує на функцію або функтор за допомогою якого буде проводитисбь порівняння. Функція mismatch повертає пару вхідних ітераторів, перший з яких вказує на першу позицію, що не співпадає в першому контейнері, а другий — в другому.

Приклад:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::mismatch
#include <vector>       // std::vector
#include <utility>      // std::pair

// порівняння лише 2 останніх розрядів
bool last2DigitsCmp (int i, int j) {
  return (i%100 == j % 100);
}

int main () {
  int mydata[] = {10, 20, 30, 40, 50};
  std::vector<int> myvector(mydata,mydata + 5);// myvector: 10
20 30 40 50

  int myints[] = {10,120,130,320,1024};                    //
myints: 10 120 130 320 1024

  std::pair<std::vector<int>::iterator,int*> mypair;

  // порівняння звичайною рівністю:
  mypair = std::mismatch (myvector.begin(), myvector.end(),
myints);
  std::cout << "Mismatching pair: " << *mypair.first;
  std::cout << " , " << *mypair.second << '\n';

  ++mypair.first; ++mypair.second;

  // using predicate comparison:
  mypair = std::mismatch (mypair.first, myvector.end(),
mypair.second, last2DigitsCmp);
  std::cout << "2nd mismatching 2 digits pair(): " <<
*mypair.first;
  std::cout << " , " << *mypair.second << '\n';
}
```

Результат:

```
Mismatching pair: 20 , 120
```

```
2nd mismatching 2 digits pair(): 40 , 320
```

Функція equal

Функція equal на відміну від mismatch визначає чи співпадає вміст двох послідовностей.

```
// equal algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::equal
#include <vector>       // std::vector

bool mypredicate (int i, int j) {
  return (i==j);
}

int main () {
  int myints[] = {20,40,60,80,100};               //   myints:
20 40 60 80 100
  std::vector<int>myvector (myints,myints+5);     // myvector:
20 40 60 80 100

  // using default comparison:
  if ( std::equal (myvector.begin(), myvector.end(), myints) )
    std::cout << "The contents of both sequences are equal.\n";
  else
    std::cout << "The contents of both sequences differ.\n";

  myvector[3]=81;                                 // myvector:
20 40 60 81 100

  // using predicate comparison:
  if ( std::equal (myvector.begin(), myvector.end(), myints,
mypredicate) )
    std::cout << "The contents of both sequences are equal.\n";
  else
    std::cout << "The contents of both sequences differ.\n";
}
```

Функції search, search_n

Функції search, search_n — на відміну від функцій пошуку find та find_if, шукають не лише єдиний елемент, а діапазон чи колекцію та видають результат у вигляді однонаправленого ітератору на перший знайдений елемент, або на кінцевий елемент, якщо цих елементів не було знайдено.

Інтерфейс функції: search (pos1_1,pos1_2, pos2_1, pos2_2) або Інтерфейс функції: search (pos1_1,pos1_2, pos2_1, pos2_2, predicate)

Функція search_n з інтерфейсом search_n (where_pos1,where_pos2, count, val) або search_n (where_pos1,where_pos2, count, val, predicat) шукає чи входить в даний діапазон count значень val (порівняння може відбуватись по функції у предикаті predicat). Якщо входить, то вона повертає ітератор на перше входження, інакше ітератор на кінець.

Приклад:

```
#include <bits/stdc++.h>
bool mypredicate (int i, int j) {
  return (i%2==j%2); // елементи рівні, якщо мають однакову парність
}

int main () {
  std::vector<int> haystack;
  // встановлюємо значення:        haystack: 10 20 30 40 50 60 70 80 90
  for (int i=1; i<10; i++) haystack.push_back(i*10);
  // шукаємо з предикатом по замовченню
  int needle1[] = {40,50,60,70,60,60};
  std::vector<int>::iterator it;
  it = std::search (haystack.begin(), haystack.end(), needle1,needle1+4);
  if (it!=haystack.end())
    std::cout << "needle1 found at position " << (it-haystack.begin()) << '\n';
  else
    std::cout << "needle1 not found\n";

 // пошук використовуючи власний предикат
  int needle2[] = {20,30,50};
  it = std::search (haystack.begin(), haystack.end(), needle2, needle2+3, mypredicate);

  if (it!=haystack.end())
    std::cout << "needle2 equal parity at position " << (it-haystack.begin()) << '\n';
  else
    std::cout << "needle2 equal parity not found\n";
```

```
  haystack.push_back(90);  haystack.push_back(80);
  it = std::search_n (haystack.begin(), haystack.end(), 2,90);
  if (it!=haystack.end())
    std::cout << "2 90th are found at position " << (it-
haystack.begin()) << '\n';
  else
    std::cout << "2 90th not found\n";

}
```
Результат:
```
needle1 found at position 3
needle2 equal parity at position 0
2 90th are found at position 8
```

Модифікуючі операції з послідовностями (Modifying sequence operations)

Функції копіювання copy та copy_backward

Функція copy призначена для копіювання діапазону значень з колекції у іншу колекцію. Інтерфейс функції copy (pose1, pos2, result_pos), остання позиція вказує за яку позицію потрібно копіювати. Функція copy_backward — яке відрізняється тим, що останній параметр вказує не на початок на кінець результату. В C++11 додали ще форми copy_n, яка копіює n чисел з заданого місця та copy_if, де вказується умова копіювання (результатом функції є вказівник на наступний після скопійованого блоку елемент).
Зокрема, метод copy з використанням ostream_iterator дозволяэ записати ще одну форму виведення булдь-якої колекції:
std::copy (s.begin(), s.end(), std::ostream_iterator<T>(std::cout, " ,"));

Приклад (з використанням C++11):
```
// use c++11
#include <iostream>     // std::cout, ostream_iterator
#include <algorithm>    // std::copy
#include <vector>       // std::vector
#include <iterator>     //ostream_iterator

template<class V,typename T> void printCont(const V & s, T val){
   std::copy                (s.begin(),                s.end(),
std::ostream_iterator<T>(std::cout, " ,"));
   std::cout<<"\n";
}
int main () {
  int myints[]={10,20,30,40,50,60,70};
  std::vector<int> myvector (7);
  std::vector<int> myvector2(8);
```

```
  std::copy( myints, myints+7, myvector.begin() );
  std::cout << "myvector contains:";
  printCont(myvector, myvector.at(0));
  std::cout<<"\n";
  std::copy_backward(    myvector.begin()+3,    myvector.end(),
myvector.end()-3 );
  printCont(myvector, myvector.at(0));

  std::copy_n( myvector.begin()+3, 3, myvector2.begin());
  printCont(myvector2, myvector2.at(0));
  auto it = std::copy_if (myvector.begin(), myvector.end()-5,
                          myvector2.begin()+3, [](int x){return
x%20==0;} );
                          // використовуємо лямбду – остача від
ділення на 20
  printCont(myvector2, myvector2.at(0));
  it--;
  std::cout<<"*it="<<*it;
}
```

Результат:

```
myvector contains:10 ,20 ,30 ,40 ,50 ,60 ,70 ,

40 ,50 ,60 ,70 ,50 ,60 ,70 ,
70 ,50 ,60 ,0 ,0 ,0 ,0 ,0 ,
70 ,50 ,60 ,40 ,0 ,0 ,0 ,0 ,
*it=40
```

Функції заміни: replace, replace_if, replace_copy, replace_copy_if

Методи replace, replace_if, replace_copy, replace_copy_if змінюють вказане значення в колекції (у формах replace_copy, replace_copy_if одночасно модифікується і те значення яким було модифіковано – тобто відбувається обмін значеннями).

```
// replace_copy example
#include <iostream>     // std::cout
#include <algorithm>    // std::replace_copy
#include <vector>       // std::vector
#include <cmath>

// друкує колекцію T
template <class T>
void printCollection(const T &v){
```

```
  for (typename T::const_iterator it=v.begin(); it!=v.end();
++it){
    std::cout << " " << *it;
  }
  std::cout << "\n";
}

bool isSqr(int x){ // Істина, якщо x - повний квадрат
   int p = round(sqrt(x));
   return p*p == x;
}

int main () {
  int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };

  std::vector<int> myvector (8);
  std::replace_copy (myints, myints+8, myvector.begin(), 20,
99);

  std::cout << "myvector contains:";
  /*for (std::vector<int>::iterator it=myvector.begin();
it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';*/
  printCollection(myvector);
  std::vector<int> v1,v2;

  // set some values:
  for (int i=1; i<10; i++) v1.push_back(i);          // 1 2 3
4 5 6 7 8 9

  v2.resize(v1.size());   // allocate space
  std::replace_copy_if (v1.begin(), v1.end(), v2.begin(),
isSqr, 0);
                                                      // 0 2 3
0 5 6 7 8 0

  std::cout << "v2 contains:";
  printCollection(v2);
}
```

Результат:

```
myvector contains: 10 99 30 30 99 10 10 99
```

```
v2 contains: 0 2 3 0 5 6 7 8 0
```

Функції видалення remove, remove_if, remove_copy, remove_copy_if
Методи remove, remove_if, remove_copy, remove_copy_if — дозволяють видаляти елементи за значенням або по значенню предикату.

```
#include <iostream>     // std::cout
#include <algorithm>    // std::remove_if
#include <vector>

bool IsPower2 (unsigned i) { // чи є число ступенем двійки
  return (i & (i-1)) == 0;
}

template <class T>   // друк значень в заданому інтервалі
void printRange(T a, T b){
     std::cout << "the range contains:";
     while(a!=b){
          std::cout<<*a<<", ";
          ++a;
     }
  std::cout << '\n';
}

// друкує колекцію T
template <class T>
void printCollection(const T &v){
  for (typename T::const_iterator it=v.begin(); it!=v.end();
++it){
    std::cout << " " << *it;
  }
  std::cout << "\n";
}

int main () {
  unsigned myints[] = {1,2,3,4,5,6,7,8,9};            // 1 2 3
4 5 6 7 8 9

  // задаємо інтервал вказівниками
  unsigned* pbegin = myints;                          //
вказівник на 1
  unsigned* pend = myints+sizeof(myints)/sizeof(int); //
вказівник після 9
```

```
  pend = std::remove_if (pbegin, pend, IsPower2);   // 3, 5,
6, 7, 9
  printRange(pbegin, pend);

  unsigned myints2[] = {10,20,30,30,20,10,10,20};
  // задаємо інтервал
  pbegin = myints2;
  pend = myints2+sizeof(myints2)/sizeof(unsigned);
  pend = std::remove(pbegin, pend, 20);         // 10, 30, 30,
10, 10,

  printRange(pbegin, pend);
  unsigned myints3[] = {1,2,3,4,5,6,7,8,9};
  std::vector<unsigned> myvector (9);
  std::vector<unsigned>::iterator vend = std::remove_copy_if
(myints3,myints3+9,myvector.begin(), IsPower2);
  // 3, 5, 6, 7, 9, 6, 7, 9, 0,
  printRange(myvector.begin(), vend);
  printRange(myvector.begin(), myvector.end());

}
```

Результат:

```
the range contains:3, 5, 6, 7, 9,
the range contains:10, 30, 30, 10, 10,
the range contains:3, 5, 6, 7, 9,
the range contains:3, 5, 6, 7, 9, 0, 0, 0, 0,
```

Примітка. Звертаємо увагу на різницю результатів в останніх викликах printRange(myvector.begin(), vend) та printRange(myvector.begin(), myvector.end()). Тобто насправді remove видаляє елементи не з колекції, а з інтервалу, який ми вказали у якості аргументів та повертає правий кінець цього інтервалу! Для того щоб реально видалити елементи з колекції можна скористатись ідеомою erase та remove:

Приклад:

```
  std::copy(myints3,myints3+9,myvector.begin());
  printRange(myvector.begin(), myvector.end());
  myvector.erase(std::remove_if(myvector.begin(),
myvector.end(), IsPower2),myvector.end());
  printRange(myvector.begin(), myvector.end());
```

Результат:

```
the range contains:1, 2, 3, 4, 5, 6, 7, 8, 9,
the range contains:3, 5, 6, 7, 9,
```

Функції видалення дублікатів unique та unique_copy
Функції unique та unique_copy дозволяють видаляти послідовні однакові значення в послідовності

Функції обміну: swap та swap_ranges, iter_swap
Функції swap та swap_ranges, iter_swap здійснюють обмін колекціями або діапазонів значень або значеннями ітераторів.

Функція перетворення: transform
Метод transform застосовує дану функцію до колекції(діапазону) або двох колекцій(діапазонів) та модифікує при цьому одну з них.

```
// transform algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::transform
#include <vector>       // std::vector
#include <functional>   // std::plus

int increment(int i) {
 return ++i;
}

int add(int x, int y){
   return x+y;
}

int main () {
  int mas[]={1,10,100,1000,10000 };
  std::vector<int> v1(mas,mas+5);// v1: 1,10,100,1000,10000
  std::vector<int> v2;

  v2.resize(v1.size());                              // allocate
space

  std::transform (v1.begin(), v1.end(), v2.begin(), increment);
                                                    //        v2:
2,11,101,1001,10001
  std::cout<<"v2 =";
  for (std::vector<int>::iterator it=v2.begin(); it!=v2.end();
++it)
    std::cout << " " << *it;
  std::cout << "\n";
```

```
  // std::plus додає два елементи:
  std::transform (v1.begin(), v1.end(), v2.begin(), v1.begin(),
add);
                                                  // v1: 21 41
61 81 101

  std::cout << "v1 =";
  for (std::vector<int>::iterator it=v1.begin(); it!=v1.end();
++it)
    std::cout << " " << *it;
  std::cout << "\n";
}
```
Результат:
```
v2 = 2 11 101 1001 10001
v1 = 3 21 201 2001 20001
```

Функції заповнення fill, fill_n та generate, generate_n

Методи fill та fill_n дозволяють швидко ініціалізувати колекцію даними.
```
 std::vector<int> myvector (8,10);        // myvector: 10 10 10
10 10 10 10 10

  std::fill_n (myvector.begin(),4,20);      // myvector: 20 20
20 20 10 10 10 10
  std::fill_n (myvector.begin()+3,3,33);    // myvector: 20 20
20 33 33 33 10 10

  std::cout << "myvector contains:";
  for     (std::vector<int>::iterator     it=myvector.begin();
it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  std::fill  (myvector.begin(),myvector.begin()+4,5);       //
myvector: 5 5 5 5 0 0 0 0
  std::fill  (myvector.begin()+3,myvector.end()-2,8);       //
myvector: 5 5 5 8 8 8 0 0
```

Методи generate та generate_n дозволяють ініціалізувати чи модифікувати колекцію за допомогою даної функції.

```
// generate algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::generate
#include <vector>       // std::vector
#include <ctime>        // std::time
#include <cstdlib>      // std::rand, std::srand

// function generator:
int RandomNumber () { return (std::rand()%100); }

// class generator:
struct c_unique {
  int current;
  c_unique() {current=0;}
  int operator()() {return ++current;}
} UniqueNumber;

int main () {
  std::srand ( unsigned ( std::time(0) ) );

  std::vector<int> v1 (8);

  std::generate (v1.begin(), v1.end(), RandomNumber);

  std::cout << "v1 random contains:";
  printCollection(v1);

  std::generate (v1.begin(), v1.end(), UniqueNumber);

  std::cout << "v1 unique contains:";
  printCollection(v1);
}
```

Результат:

```
v1 random contains: 91 81 2 90 27 46 69 64
v1 unique contains: 1 2 3 4 5 6 7 8
```

Функції перестановок reverse та reverse_copy, rotate та rotate_copy, random_shuffle

Методи reverse та reverse_copy — дозволяють інвертувати колекцію
Методи rotate та rotate_copy — роблять циклічний зсув колекції
Метод random_shuffle — випадковим чином перемішує послідовність.

З С++11 додали також функцію shuffle, яка перемішує послідовність за допомогою даного генератору псевдовипадкових чисел

Операції розділення (partitions)

Функція partition

Функція partition (pos1,pos2, pred) модифікує діапазон [pos1,pos2) таким чином, що елменти для яких предикат pred повертає true передують тим, де він повертає false. Результат — ітератор, що вказує на перший елемент, який повертає false в новому діапазоні.
Відносний порядок при цьому не зобов'язаний зберігатися, якщо його потрібно зберегти використовується функція stable_partition.
Приклад:

```
// partition algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::partition
#include <vector>       // std::vector

bool isOdd (unsigned i) {
 return i & 1;
}

template <class T>
void printRangeVector(const T start, const T end ){
  for (T it=start; it!=end; ++it){
    std::cout << " " << *it;
  }
  std::cout << "\n";
}

int main () {
  std::vector<unsigned> myvector;
  // set some values:
  for (unsigned i=1; i<10; ++i) myvector.push_back(i); // 1 2 3
4 5 6 7 8 9

  std::vector<unsigned>::iterator bound;
  bound  =  std::partition  (myvector.begin(),  myvector.end(),
isOdd);

  // print out content:
```

```
  std::cout << "odd elements:";
  printRangeVector(myvector.begin(), bound);

  std::cout << "even elements:";
  printRangeVector(bound,myvector.end());
}
```
Результат:
```
odd elements: 1 9 3 7 5
even elements: 6 4 8 2
```

Якщо в тому ж самому коды поміняти partition на stable_partition, то порядок збережеться:
```
 std::vector<unsigned> myvector;
  // set some values:
  for (unsigned i=1; i<10; ++i) myvector.push_back(i); // 1 2 3
4 5 6 7 8 9

  std::vector<unsigned>::iterator bound;
  bound     =     std::stable_partition     (myvector.begin(),
myvector.end(), isOdd);

  // print out content:
  std::cout << "odd elements:";
  printRangeVector(myvector.begin(), bound);

  std::cout << "even elements:";
  printRangeVector(bound,myvector.end());
```

Результат:
```
odd elements: 1 3 5 7 9
even elements: 2 4 6 8
```

Примітка. З C++11 додали також функції is_partitioned, partition_copy, partition_point

Операції сортування (Sorting)

Функція sort

Функція sort — сортує за зростанням вказаний інтервал за вказаним бінарним предикатом (за замовченням — це стандартний або перевантажений оператор <)

stable_sort — працює майже так саме як sort, але при цьому обов’язково зберігається взаємний порядок елементів, для яких критерій порівняння визначав еквівалентність.

```
// stable_sort example
#include <iostream>     // std::cout
#include <algorithm>    // std::stable_sort
#include <vector>       // std::vector

// друкує колекцію T
template <class T>
void printCollection(const T &v){
  for (typename T::const_iterator it=v.begin(); it!=v.end();
++it){
    std::cout << " " << *it;
  }
  std::cout << "\n";
}

bool compare_as_ints (double i,double j){
  return (int(i)<int(j));
}

int main () {
  double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32,
1.62, 2.58};

  std::vector<double> myvector;

  myvector.assign(mydoubles,mydoubles+8);

  std::cout << "using default comparison:";
  std::stable_sort (myvector.begin(), myvector.end());
  printCollection(myvector);

  myvector.assign(mydoubles,mydoubles+8);
  std::cout << "using 'compare_as_ints' :";
  std::stable_sort (myvector.begin(), myvector.end(),
compare_as_ints);
  printCollection(myvector);
}
```

Результат:

```
using default comparison: 1.32 1.41 1.62 1.73 2.58 2.72 3.14
4.67
using 'compare_as_ints' : 1.41 1.73 1.32 1.62 2.72 2.58 3.14
4.67
```

Функція partial_sort та partial_sort_copy

Функція часткового сортування partial_sort (pos1,middle,pos2) та partial_sort (pos1, middle, pos2, comp) — сортує елементи діапазону таким чином, що елементи перед *middle* стають найменшими елементами діапазону та відсортовані за неспаданням.
Критерій порівняння - operator< для першої версії та бінарний предикат *comp* для другої.

```
// partial_sort example
#include <iostream>     // std::cout
#include <algorithm>    // std::partial_sort
#include <vector>       // std::vector

bool myfunction (int i,int j) { return (i<j); }

int main () {
  int myints[] = {9,8,7,6,5,4,3,2,1};
  std::vector<int> myvector (myints, myints+9);

  // using default comparison (operator <):
  std::partial_sort   (myvector.begin(),   myvector.begin()+5,
myvector.end());

  // using function as comp
  std::partial_sort   (myvector.begin(),   myvector.begin()+5,
myvector.end(),myfunction);

  // print out content:
  std::cout << "myvector contains:";
  for     (std::vector<int>::iterator     it=myvector.begin();
it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
}
```

Результат:

```
myvector contains: 1 2 3 4 5 9 8 7 6
```

Функція partial_sort_copy працює так саме як partial_sort, але зберігає невідсортовану частину.

Функція nth_element

Функція nth_element(pos1,middle,pos2) модифікує діапазон таким чином, що елемент на позиції middle займає ту позицію, яка в нього була би в відсортованому масиві.
Приклад:

```
// nth_element example
#include <iostream>     // std::cout
#include <algorithm>   // std::nth_element, std::random_shuffle
#include <vector>       // std::vector

// друкує колекцію T
template <class T>
void printCollection(const T &v){
  for (typename T::const_iterator it=v.begin(); it!=v.end();
++it){
    std::cout << " " << *it;
  }
  std::cout << "\n";
}

bool my_comparison (int i,int j) { return (i>j); }

int main () {
  int mas[] = {1,2,3,4,5,6,7,8,9};
  std::vector<int> myvector(mas,mas+9);

  std::random_shuffle (myvector.begin(), myvector.end());
  std::cout<<"v=";
  printCollection(myvector);
  // використвоуючи порівняння(operator <):
  std::nth_element                          (myvector.begin(),
myvector.begin()+5,myvector.end());
  std::cout<<"v (v[5]=6):";
  printCollection(myvector);
  // використовуючи порівняння my_comparison

std::nth_element(myvector.begin(),myvector.begin()+5,myvector.
end(),my_comparison);
 std::cout<<"v v[5] =4:";
```

```
  printCollection(myvector);
}
```
Результат:
```
v= 5 4 8 9 1 6 3 2 7
v (v[5]=6): 4 2 3 1 5 6 7 8 9
v v[5] =4: 5 9 8 7 6 4 3 2 1
```

З C++11 додали функції is_sorted, is_sorted_until — які визначають чи відсортований даний діапазон або чи відсортований він до деякого елементу.

Бінарний пошук

Алгоритми бінарного пошуку працюють на відсортованих або частково відсортованих колекціях

Функція binary_search
Ця функція виконує бінарний пошук даного елементу в послідовності.
Приклад:
```
#include <iostream>     // std::cout
#include <algorithm>    // std::binary_search, std::sort
#include <vector>       // std::vector
#include <iterator>     //
using namespace std;

bool Comparator (int i,int j) { return (i<j); }

int main () {
const int N = 8;
int a2[N] ={1234,5432,8943,3346,9831,7842,8863,9820};
cout << "Before sorting:";
copy(a2, a2+N, ostream_iterator<int>(cout,  " "));
cout << endl;
sort (a2, a2+N, Comparator);
cout << "After sorting in descending order:";
copy(a2, a2+N, ostream_iterator<int>(cout,  " "));
cout << endl;


if (binary_search(a2, a2 + N, 2, Comparator))   // Comparator()
- obligatory!!!
 cout << "Element found in the array";
```

```
else
 cout << "Element not found in the array";

copy(a2, a2+N, ostream_iterator<int>(cout,  " "));


reverse(a2,a2+N); // or search with reverse
cout<<"\n";

copy(a2, a2+N, ostream_iterator<int>(cout,  " "));

if (binary_search(a2, a2+N, 7842))
  cout << "Element found in the array";
else
  cout << "Element not found in the array";
}

Before sorting:1234 5432 8943 3346 9831 7842 8863 9820
After sorting in descending order:1234 3346 5432 7842 8863 8943
9820 9831
Element not found in the array1234 3346 5432 7842 8863 8943 9820
9831
9831 9820 8943 8863 7842 5432 3346 1234 Element not found in the
array
```

Функція equal_range

Функція equal_range(pos1,pos2, val) повертає піддіапазон тих елементів діапазону (як пару на початок та кінець піддіапазону), що еквівалентні val.

Приклад:

```
// equal_range example
#include <iostream>     // std::cout
#include <algorithm>    // std::equal_range, std::sort
#include <vector>       // std::vector

bool mygreater (int i,int j) { return (i>j); }

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  std::vector<int> v(myints,myints+8);                        //
10 20 30 30 20 10 10 20
```

```
std::pair<std::vector<int>::iterator,std::vector<int>::iterato
r> bounds;

  // using default comparison:
  std::sort (v.begin(), v.end());                              //
10 10 10 20 20 20 30 30
  bounds=std::equal_range (v.begin(), v.end(), 20);            //
^        ^

  // using "mygreater" as comp:
  std::sort (v.begin(), v.end(), mygreater);                   //
30 30 20 20 20 10 10 10
  bounds=std::equal_range (v.begin(), v.end(), 20, mygreater);
//       ^        ^

  std::cout << "bounds at positions " << (bounds.first -
v.begin());
  std::cout << " and " << (bounds.second - v.begin()) << '\n';
}
Результат:
bounds at positions 2 and 5
```

Функції lower_bound, upper_bound

Функції lower_bound(pos1,pos2, val), upper_bound(pos1,pos2, val) повертають відповідно ітератори на перший та останній елемент піддіапазону еквівалентних val значень.

Приклад:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::lower_bound, std::upper_bound,
std::sort
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  std::vector<int> v(myints,myints+8);           // 10 20 30 30
20 10 10 20

  std::sort(v.begin(), v.end());                 // 10 10 10 20
20 20 30 30
```

```
  std::vector<int>::iterator low,up;
  low=std::lower_bound (v.begin(), v.end(), 20); //          ^
  up=   std::upper_bound   (v.begin(),   v.end(),   20);   //
^

  std::cout << "lower_bound at position " << (low- v.begin())
<< '\n';
  std::cout << "upper_bound at position " << (up - v.begin())
<< '\n';
}
```

Результат:

```
lower_bound at position 3
upper_bound at position 6
```

Функції злиття

Дані функції працюють для відсортованих послідовностей.

Функції merge та inplace_merge

Функція merge виконує алгоритм злиття двох відсортованих послідовностей, а функція inplace_merge виконує злиття діапазонів [first,middle) та [middle,last), поміщуючи результат в діапазон [first,last).
Метод зберігає відносне розташування еквівалентних елементів

Приклад:

```
#include <iostream>     // std::cout
#include  <algorithm>       //  std::inplace_merge,  std::sort,
std::copy
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);
  std::vector<int>::iterator it;

  std::sort (first,first+5);
  std::sort (second,second+5);

  it=std::copy (first, first+5, v.begin());
     std::copy (second,second+5,it);

  std::inplace_merge (v.begin(),v.begin()+5,v.end());
```

```
  std::cout << "The resulting vector contains:";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
}
```
Результат:
```
The resulting vector contains: 5 10 10 15 20 20 25 30 40 50
```

Функція includes

Ця функція перевіряє, чи входить одна відсортована послідовність в іншу.

Приклад:
```
#include <iostream>     // std::cout
#include <algorithm>    // std::includes, std::sort
#include <cstring>
#include <cctype>

bool myCharCmp (char a, char b) {
  return islower(a)<islower(b);
}

int main () {
  char long_word[] = "abracaDABRA";
  char small_word[] = "cadar";

  std::sort (long_word,long_word+strlen(long_word));
  std::sort (small_word,small_word+strlen(small_word));

  // using default comparison:
if    (std::includes(long_word,    long_word+strlen(long_word),
small_word, small_word+strlen(small_word)) ){
    std::cout << "abracaDABRA includes cadar!\n";
}
  else{
     std::cout << "abracaDABRA do not includes cadar!\n";
  }

  // using myfunction as comp:
if   (std::includes(long_word,   long_word+strlen(long_word)   ,
small_word, small_word+strlen(small_word), myCharCmp)  )
    std::cout << "abracadara includes cadar!\n";
```

```
}
```
Результат:
```
abracaDABRA do not includes cadar!
abracadara includes cadar
```

Функції роботи з множинами set_union, set_intersection, set_difference, set_symmetric_difference

Функції set_union, set_intersection, set_difference, set_symmetric_difference - виконують відповідно об'єднання, перетин, різницю та симетричну різницю двох відсортованих послідовностей.

Приклад:

```
#include <iostream>     // std::cout
#include  <algorithm>       //  std::set_symmetric_difference, std::sort
#include <vector>       // std::vector
#include <set>

// друкує колекцію T
template <class T>
void printCollection(const T &v){
  for (typename T::const_iterator it=v.begin(); it!=v.end(); ++it){
    std::cout << " " << *it;
  }
  std::cout << "\n";
}

int main () {
  int col1[] = {5,10,15,20,25};
  int mas2[] = {50,40,30,20,10};
  std::vector<int> v1,v2,v3;                    // 0  0  0  0  0  0  0  0  0  0
  v1.resize(10);v2.resize(5);v3.resize(10);
  std::set<int> col2(mas2, mas2+5);
  //std::vector<int>::iterator it;

  std::sort (col1,col1+5);     //  5 10 15 20 25
  //std::sort (second,second+5);   // 10 20 30 40 50

  std::set_symmetric_difference (col1, col1+5, col2.begin(), col2.end(), v1.begin());
```

```
  std::cout<<"symmetric difference v1 = col1, coll2:";
  printCollection(v1);

  std::set_difference (col1, col1+5, col2.begin(), col2.end(),
v2.begin());
  std::cout<<"difference (5 items - 2 are extra) v2:";
  printCollection(v2);

  std::vector<int>::iterator it =std::set_union (col1, col1+5,
col2.begin(), col2.end(), v3.begin());
  v3.erase(it,v3.end()); // remove extra items
  std::cout<<"union v3:";
  printCollection(v3);

  it = std::set_intersection(v1.begin(), v1.end(), v3.begin(),
v3.end(), v1.begin());
  v1.erase(it,v1.end()); //// remove extra items
  std::cout<<"intersecton v1,v3:";
  printCollection(v1);
}
```

Результат:

```
symmetric difference v1 = col1, coll2: 5 15 25 30 40 50 0 0 0 0
difference (5 items - 2 are extra) v2: 5 15 25 0 0
union v3: 5 10 15 20 25 30 40 50
intersecton v1,v3: 5 15 25 30 40 50
```

Мінімум/Максимум

Функції min, max — повертають відповідно значення мінімально та максимального елементу послідовності

Функції min_element, max_element - повертають відповідно однонаправлений ітератор на мінімальний та максимальний елементи послідовності

З C++11 додали також функції:
Функції minmax, minmax_element — які повертають відповідно одночасно значення мінімального та максимального елементів або однонаправлений ітератор на них у вигляді пари.

Приклад (C++11):

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
```

```
int main() {
     int arr[] = {10, 20, 5, 23 ,42 , 15};
     int n = sizeof(arr)/sizeof(arr[0]);
     vector<int> vect(arr, arr+n);
     //  реверс вектору
     reverse(vect.begin(), vect.end());
          // виведення вектору
     cout << "Vector after reversing is: ";
     for (int i=0; i<vect.size(); i++)
          cout << vect[i] << " ";
       // тестуємо функції пошуку мінімума та максимума
     cout << "\nMaximum element of vector is: ";
     cout << *max_element(vect.begin(), vect.end())<<" or " <<
max({10, 20, 5, 23 ,42 , 15});
     cout << "\nMinimum element of vector is: ";
     cout << *min_element(vect.begin(), vect.end())<<" or " <<
min({10, 20, 5, 23 ,42 , 15});

cout <<"\nmin:"<< *minmax_element(vect.begin(),
vect.end()).first <<" max=" << minmax({10, 20, 5, 23 ,42 ,
15}).second;
}
```

Результат:

```
Vector after reversing is: 15 42 23 5 20 10
Maximum element of vector is: 42 or 42
Minimum element of vector is: 5 or 5
min:5 max=42
```

Робота з купою

Функції make_heap, push_heap, pop_heap, sort_heap – відповідно створюють структуру даних купа, додають елемент до купи, видаляють елемент з купи та сортують її.

```
// range heap example
#include <iostream>     // std::cout
#include <algorithm>    // std::make_heap, std::pop_heap, std::push_heap, std::sort_heap
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,5,15};
  std::vector<int> v(myints,myints+5);

  std::make_heap (v.begin(),v.end());
  std::cout << "initial max heap   : " << v.front() << '\n';

  std::pop_heap (v.begin(),v.end()); v.pop_back();
  std::cout << "max heap after pop : " << v.front() << '\n';
```

```
  v.push_back(99); std::push_heap (v.begin(),v.end());
  std::cout << "max heap after push: " << v.front() << '\n';

  std::sort_heap (v.begin(),v.end());

  std::cout << "final sorted range :";
  for (unsigned i=0; i<v.size(); i++)
    std::cout << ' ' << v[i];

  std::cout << '\n';

  return 0;
}
```

3 Ci++11 додали також функції is_heap(), is_heap_until

```
// is_heap example
#include <iostream>     // std::cout
#include <algorithm>    // std::is_heap, std::make_heap, std::pop_heap
#include <vector>       // std::vector

int main () {
  std::vector<int> foo {9,5,2,6,4,1,3,8,7};

  if (!std::is_heap(foo.begin(),foo.end()))
    std::make_heap(foo.begin(),foo.end());

  std::cout << "Popping out elements:";
  while (!foo.empty()) {
    std::pop_heap(foo.begin(),foo.end());   // moves largest element to back
    std::cout << ' ' << foo.back();         // prints back
    foo.pop_back();                         // pops element out of container
  }
  std::cout << '\n';

  return 0;
}
```

Інші

До інших функцій входять достатньо рідко використовувані функції lexicographical_compare (порівняння вмісту послідовностей лексикографічно), next_permutation (наступне перемішування) та prev_permutation (попереднє перемішування).

Алгоритми бібліотеки numeric

В бібліотеку numeric входять наступні функції:

Функція accumulate

Функція accumulate(pos1,pos2,init) або accumulate(pos1,pos2,init, fun) виконує послідовне (комулятивне) сумування елементів послідовності або послідовне виконання заданої функції fun від першого до останнього починаючи з деякого значення init.

Приклади:

```
#include <iostream>
```

```
#include <numeric>   // для алгоритмів

void accum1_massiv(){
     const int N = 8;
     int a[N] = {4, 12, 3, 6, 10, 7, 8, 5}, sum = 0;
     sum = accumulate(a, a+N, sum);
     std::cout << "Sum of all elements:  " << sum << "\n";
     std::cout << "1000 + a[2] + a[3] + a[4]  = "  <<
accumulate(a+2, a+5, 1000) <<"\n";
}

void accum2_massiv(){
     const int N = 4;
     int a[N] = {2, 10, 5, 3}, prod =  1;
     prod = accumulate(a, a+N, prod, multiplies<int>());
     std::cout << "Product of all elements: " << prod << endl;
}

int main(){
  accum1_massiv();
  accum2_massiv();
}
```

Результат:

```
Sum of all elements:  55
1000 + a[2] + a[3] + a[4]  = 1019
Product of all elements: 300
```

Функція adjacent_difference

Якщо x — це елемент [first,last) та y — елемент в result, то результат цієї функції буде наступний:
y0 = x0
y1 = x1 - x0
y2 = x2 - x1
y3 = x3 - x2
y4 = x4 - x3

Функція inner_product

Функція рахує комулятивний скалярний добуток інтервалу *init* де добуток рахується для пари на яку вказують числа з *first1* та *first2*. Дві операції за замовченням (додавання для результатів та множення для пар) можуть бути перевантажені бінарними функціями *binary_op1* та *binary_op2*.

Функція partial_sum

Якщо *x* представляє елемент в [first,last), а *y* представляє елемент в *result*, то результат функції може бути представлений:
y0 = x0
y1 = x0 + x1
y2 = x0 + x1 + x2
y3 = x0 + x1 + x2 + x3
y4 = x0 + x1 + x2 + x3 + x4

Приклади:

```
#include <iostream>     // std::cout
#include <functional>   // std::multiplies,  divides
#include  <numeric>              //  std::adjacent_difference,
inner_product,  partial_sum

int myop (int x, int y) {return x^y;}
int myaccumulator (int x, int y) {return x-y;}
int main () {
  int val[] = {1,2,3,5,9,11,12};
  int result[7];

  std::adjacent_difference (val, val+7, result);
  std::cout << "using default adjacent_difference: ";
  for (int i=0; i<7; i++) std::cout << result[i] << ' ';
  std::cout << '\n';

  std::adjacent_difference      (val,      val+7,      result,
std::multiplies<int>());
  std::cout << "using functional operation multiplies: ";
  for (int i=0; i<7; i++) std::cout << result[i] << ' ';
  std::cout << '\n';

  std::adjacent_difference (val, val+7, result, myop);
  std::cout << "using custom function: ";
  for (int i=0; i<7; i++) std::cout << result[i] << ' ';
  std::cout << '\n';

  int init = 100;
  int series1[] = {10,20,30};
  int series2[] = {1,2,3};

  std::cout << "using default inner_product: ";
```

```
  std::cout                                                    <<
std::inner_product(series1,series1+3,series2,init);
  std::cout << '\n';

  std::cout << "using functional operations: ";
  std::cout                                                    <<
std::inner_product(series1,series1+3,series2,init,

std::minus<int>(),std::divides<int>());
  std::cout << '\n';

  std::cout << "using custom functions: ";
  std::cout                                                    <<
std::inner_product(series1,series1+3,series2,init,
                                    myaccumulator,myop);
  std::cout << '\n';

  //int val[] = {1,2,3,4,5};
 // int result[5];

  std::partial_sum (val, val+5, result);
  std::cout << "using default partial_sum: ";
  for (int i=0; i<5; i++) std::cout << result[i] << ' ';
  std::cout << '\n';

  std::partial_sum         (val,         val+5,         result,
std::multiplies<int>());
  std::cout << "using functional operation multiplies: ";
  for (int i=0; i<5; i++) std::cout << result[i] << ' ';
  std::cout << '\n';

  std::partial_sum (val, val+5, result, myop);
  std::cout << "using custom function: ";
  for (int i=0; i<5; i++) std::cout << result[i] << ' ';
  std::cout << '\n';

}
```

Результат:

```
using default adjacent_difference: 1 1 1 2 4 2 1
using functional operation multiplies: 1 2 6 15 45 99 132
using custom function: 1 3 1 6 12 2 7
using default inner_product: 240
```

```
using functional operations: 70
using custom functions: 38
using default partial_sum: 1 3 6 11 20
using functional operation multiplies: 1 2 6 30 270
using custom function: 1 3 0 5 12
```

Функція iota (C++11)

В C++11 до цього переліку функцій також додали функцію iota, яка за даним інтервалом [first,last) повертає послідовні значення *val*, які отримуються послідовним застосуванням інкременту ++val до кожного елементу.

Приклад:

```
#include <iostream>     // std::cout
#include <numeric>      // std::iota

int main () {
  int numbers[10];
  std::iota (numbers,numbers+10,100);
  std::cout << "numbers:";
  for (int& i:numbers) std::cout << ' ' << i;
  std::cout << '\n';
}
```

Результат:

```
numbers: 100 101 102 103 104 105 106 107 108 109
```

Функтори та предикати

Багато з розглянутих алгоритмів C++ мають у якості аргументу функцію, наприклад, for_each, generate, find_if.
Крім того, функція може бути аргументом конструктору для шаблонів класів set, multiset, map, multimap для задання шляху сортування ключів цих класів.
Задати цю функцію можна як безпосередньо функцію, тобто приписавши реалізацію або декларацію цієї функції до виклику її як аргумент функції чи конструктору, це фактично означає, що ми використовуємо вказівник на цю функцію. Але інколи було б бажано використати цю функцію як змінну або об’єкт, тобто мати її як конкретний екземпляр деякого класу.
Зокрема, розглянемо функцію, яка приймає лише один аргумент але під час виклику цієї функції нам потрібно передати, наприклад, ще параметр. Однак на C++ це неможливо, оскільки функція приймає лише один параметр. Що можна зробити? Очевидною відповіддю можуть бути глобальні змінні. Однак практика гарного кодування не виступає за використання глобальних змінних і стверджує, що їх слід використовувати лише тоді, коли немає іншої альтернативи.

На C++ для подібних модифікацій функції можна використати функтори, які також звуться функціональними об’єктами (Зверніть увагу, що функтори — це не те само що функції !!).
Функтори – це об’єкти, які можна обробляти так, ніби вони є функцією або вказівником на функції.
Функціональний об’єкт або функтор – це клас який перевантажує оператор виклику operator () наступним чином, що в коді

```
FunctionObjectType func;
func();
```

вираз func() є викликом оператору () об’єкту func, а не викликом функції func.
Тип функціонального об’єкту повинен бути визначеним наступним чином:

```
class FunctionObjectType {
  public:
  void operator() () {
    // Код функції
  }
};
```

У використання функціональних об'єктів є ряд переваг перед використанням функцій, а саме:
1. Функціональний об'єкт може мати стан. Фактично може бути два об'єкта одного і того ж функціонального типу, що знаходяться в різних станах в одне і теж час, що неможливо для звичайних функцій. Також функціональний об'єкт може забезпечити операції попередньої ініціалізації даних.
2. Кожен функціональний об'єкт має тип, а отже є можливість передати цей тип як параметр шаблону для вказівки певної поведінки. Наприклад, типи контейнерів з різними функціональними об'єктами відрізняються.
3. Об'єкти-функції часто виконуються швидше ніж вказівники на функції. Наприклад, вбудуване (inline) звернення до оператора () класу працює швидше, ніж функція, передана за вказівником.
Функтори найчастіше використовуються разом із STL у такому сценарії:

```
#include <iostream>
#include <set>
#include <cmath>

// друкує колекцію T
template <class T>
void printCollection(const T &v){
  for (typename T::const_iterator it=v.begin(); it!=v.end();
++it){
    std::cout << " " << *it;
```

```
  }
  std::cout << "\n";
}
//class Comparator
struct Classcomp {
  bool operator() (const int& lhs, const int& rhs) const {
    return std::abs(lhs)<=std::abs(rhs);
  }
};

int main (){
  int mas[]= {-3,-1,2,1,5};
  std::set<int,Classcomp> fifth_set;                    // class
as Compare
  for(int i=0;i<5;++i){
    fifth_set.insert(mas[i]);
  }
  printCollection(fifth_set);
}
```

Результат:

```
 1 -1 2 -3 5
```

Таким чином, функтор (або функціональний об’єкт) – це клас C++, який діє як функція. Функтори викликаються з використанням синтаксису виклику функції. Для того, щоб створити функтор, потрібно створити об'єкт MyFunctor, який перевантажує оператор “круглі дужки” (*operator()*).

Тоді код:

```
MyFunctor(10);
```

еквівалентний коду

```
MyFunctor.operator()(10);
```

Ще один приклад використання функтору — в алгоритмі transform.

```
class Increment{
     int num;
 public:
    Increment(int n): num(n) {}
    int operator()(int k){
    return num+k;
    }
};
```

Тоді рядок:
```
transform(arr, arr+n, arr, increment(to_add));
```

буде еквівалентним наступному коду:
```
 // Створення обєкту класу Increment
  Increment obj(to_add);
  // виклики оператору ()
  std::transform(arr2.begin(), arr2.end(), arr2.begin(), obj);
```

Тобто об’єкт *a* створений таким, що перевантажує *operator()*. Таким чином, функтори достатньо ефективні при використанні в C++ STL.

Предикати

Зауважимо, що частковий випадок функторів, які повертають логічний тип bool зветься предикатом. Зокрема предикати потрібні при використанні таких функцій з бібліотеки алгоритмів, як find_if(), count_if, sort тощо.
Предикати використовуються в алгоритмах сортування, пошуку, а також в усіх інших, що мають в кінці _if. Сенс у тому, що об’єкт-функція у випадку використання предикату повертає true або false у залежності від виконання необхідної умови. Це або факт відповідності об’єктом деяких властивостей, або результат порівняння двох об’єктів за визначеною ознакою.
Приклад використання предикатів.
Приклад (кількість парних елементів)
```
#include <iostream>
#include <algorithm>

class DividedByTwo{ // чи парний даний елемент
    public:
        bool operator()(const int x) const        {
            return x % 2 == 0;
        }
};

int main(){
    const std::size_t N = 5;
    int A[N] = {3, 2, 5, 6, 8};
    std::cout << std::count_if(A, A + N, DividedByTwo());
}
```
Результат:
3

Розглянемо ще один приклад використання предикату. Для того, щоб передати предикату критерій, необхідно в тілі структури створити конструктор так, як показано на прикладі.

Приклад (видаляємо з масиву числа більше за 4):

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>
// предикат-функтор повертає чи менше дане число деякого
заданого
struct Prd {
   int my_cnt;
   // Конструктор
   Prd(const int &t) : my_cnt(t) {}
   // Перегрузка операції ()
   bool operator() (const int & v) {
      //cout<<"op"<<v<<"/"<<my_cnt<<";";
      return v > my_cnt;
   }
};

int main() {
   int mas[] =  {1, 2, 3, 4, 5, 3, 7, 3, 9, 3};
   std::vector<int> num(mas, mas + 10);

   const int z = std::count_if(num.begin(), num.end(),
Prd(4)); // кількість чисел більших 4
   std::vector<int> myvector(num.size()- z); // vector розміру
кількості чисел менших 4
   //using namespace placeholders; // або рішення в стилі
c++11
   //const int z = count_if(num.begin(), num.end(),
bind(logical_not<bool>(), bind(Prd(4), _1)));
   //vector<int> myvector(z);

   //видаляємо всі числа що більше 4
   std::vector<int>::iterator it =
std::remove_copy_if(num.begin(), num.end(), myvector.begin(),
Prd(4));
   std::cout<< "\n";
```

```
   for(std::vector<int>::iterator it2 = myvector.begin();
it2!=myvector.end();++it2)
      std::cout<<*it2<<", ";
}
```
Результат:
```
1, 2, 3, 4, 3, 3, 3,
```
Зрозуміти, як працює програма можна, якщо взяти за увагу, що викликається функція, як аргумент іншої функції, тобто operator()(arg_prd, Prd(arg_crt)), де arg_prd, що передається предикату, елемент масиву, а arg_crt - аргумент-критерій методу Prd(). Але оскільки ми маємо справу з класом, то ми кажемо, що другим аргументом викликається наш конструктор. Ось у цьому й розкривається перевага функторів, у порівнянні з звичайними функціями.

Огортки функцій та стандартні функтори бібліотеки function

Починаючи зі стандарту C++11 шаблонний клас function з бібліотеки <function> є поліморфною обгорткою функцій для загального використання. Об'єкти класу function можуть зберігати, копіювати і викликати довільні об'єкти, що можуть викликатись – функції, лямбда-вирази, вирази зв'язування і інші функціональні об'єкти. Взагалі кажучи, в будь-якому місці, де необхідно використовувати вказівник на функцію для її відкладеного виклику, або для створення функції зворотного виклику, замість нього може бути використаний std :: function, який надає користувачеві велику гнучкість в реалізації.
Визначення класу:
```
template<class> class function; // undefined
template<class      R,      class...      ArgTypes>      class
function<R(ArgTypes...)>;
```
Також в стандарті C++11 визначені функції-модифікатори swap та assign і оператори порівняння (== та !=) з nullptr. Доступ до цільового об’єкту дає функція target, а до його типу — target_type. Оператор приведення function до булевого типу повертає true, коли у класу є цільовий об’єкт.
Приклад:
```
#include <iostream>
#include <functional>

struct A {
    A(int num) : num_(num){}
    void printNumberLetter(char c) const {std::cout << "Number:
" << num_  << " Letter: " << c << std::endl;}
    int num_;
};

void printLetter(char c){
```

```
    std::cout << c << std::endl;
}

struct B {
    void operator() () {std::cout << "B()" << std::endl;}
};


std::function<double(double)>                derivative(const
std::function<double(double)> &f, const double h){

  return [=](double x)->double { return (f(x+h)-f(x-h))/2/h; };
}

int main(){
    // визначаємо функтор через вказівник на функцію
    std::function<void(char)> f_print_Letter = printLetter;
    f_print_Letter('Q');

    // визначаємо функтор як лямбда-вираз
    std::function<void()> f_print_Hello = [] () {std::cout <<
"Hello world!" << std::endl;};
    f_print_Hello();

    // визначаємо функтор через зв’язувач
    std::function<void()>  f_print_Z  =  std::bind(printLetter,
'Z');
    f_print_Z();

    // визначаємо функтор як метод класу
    std::function<void(const    A&,    char)>    f_printA    =
&A::printNumberLetter;
    A a(10);
    f_printA(a, 'A');

    // присвоєння функторів
    B b;
    std::function<void()> f_B = b;
    f_B();

    // рахуємо піхідну функції як функцію
```

```
    std::function<double(double)> fd = [](double x)->double{
return x*x; } ;
    std::cout<<derivative(fd,0.001)(2.0);
}
```
Результат (1- функтор виводу літери, 2 — вивід “Hello world!” через лямбду, 3 - підставка в функтор літери Z, 4 — використання фнкутору для виведення числа та літери, 5 — присвоєння функторів, 5 — піхідна x*x в точці 2):
```
Q
Hello world!
Z
Number: 10 Letter: A
B()
4
```

Виключення bad_functional_call

Виключення типу bad_functional_call буде створено при спробі виклику функції function::operator(), якщо в неє буде відсутній цільовий об’єкт. Клас bad_functional_call є нащадком std::exception, і в нього є доступним віртуальний метод what() для отримання тексту помилки.

Приклад (C++11):
```
#include <iostream>
#include <functional>

int main(){
    std::function<void()> func = nullptr;
    try {
        func();
    } catch(const std::bad_function_call& e) {
        std::cout << e.what() << std::endl;
    }
}
```
Результат (обробиться виключення):
```
bad_function_call
```

Адаптори функторів

Більшість функторів і предикатів за кількістю операндів - бінарні, тому дуже часто доводиться використовувати функційні адаптери. Адаптери - також функтори, але вони назначені для зв’язки функторів й аргументів. До стандартних функційних адаптерів відносяться наступні:

1. bind() - зв’язує аргументи з операцією.

Шаблонна функція std::bind зветься зв’язувачем та надає підтримку часткового використання функцій. Вона прив'язує деякі аргументи до функціонального об'єкту, створюючи новий функціональний об'єкт. Тобто виклик зв’язувача відповідає виклику функціонального об'єкта з деякими певними параметрами. Передавати зв’язувачу можна або безпосередньо значення

аргументів, або спеціальні імена з простору імен std :: placeholders, які вказують зв’язувачу на те, що даний аргумент буде пов'язаний, і визначають порядок аргументів який повертається в функціональний об'єкт.

Визначення функції:

```
template<class F, class... BoundArgs>
   unspecified bind(F&& f, BoundArgs&&... bound_args);
template<class R, class F, class... BoundArgs>
   unspecified bind(F&& f, BoundArgs&&... bound_args);
```

Тут f — об’єкт виклику, bound_args — список зв’язаних аргументів. Типом що повертається є функціональний об’єкт невизначеного типу T, який може бути застосуваваним в std::function, і для якого виконується std::is_bind_expression<T>::value == true. Всередині огортка містить об’єкт типу std::decay<F>::type, побудованого з std::forward<F>(f), а також по одному об’єкту для кожного аргументу аналогичного типу std::decay<Arg_i>::type.

За допомогою bind() можна пов’язати аргументи, що викликають об’єкт безпосередньо(вказавши, наприклад, конкретне значення) або за допомогою об’єктів, що заповнюють. Щоб не згадувати цей простір імен в програмі, необхідно використати директиву:

```
using namespace placeholders;
```

У просторі імен std :: placeholders містяться спеціальні об'єкти _1, _2, ..., _N, де число N залежить від реалізації. Вони використовуються в функції bind для завдання порядку вільних аргументів. Коли такі об'єкти передаються у вигляді аргументів на функцію bind, то для них генерується функціональний об'єкт, в якому, при виклику з непов'язаними аргументами, кожен заповнювач _N буде замінений на N-й за рахунком непов'язаний аргумент. Для отримання цілого числа k з заповнювач _K передбачений допоміжний шаблонний клас std :: is_placeholder. При передачі йому заповнювач, як параметра шаблона, є можливість отримати ціле число при зверненні до його полю value. Наприклад, is_placeholder <_3> :: value поверне 3.

Приклад:

```
#include <iostream>
#include <functional>

int myPlus (int a, int b) {return a + b;}

int main(){
    std::function<int        (int)>        f(std::bind(myPlus,
std::placeholders::_1, 5));
    std::cout << f(10) << std::endl;
}
```

Результат (тут підставляється 10+5):

```
15
```

2. mem_fn() - викликає операцію, що вказує на функцію-член об’єкту;

mem_fn

Шаблоная функція std::mem_fn створює клас-огортку для вказівників на члени класу. Цей об’єкт може зберігати, копіювати та викликати член класу по вказівнику.

3. Адаптори not1 та not2 дозволяють використовувати у адаптерах заперечення відповідно для бінарних та унарних функцій.

Стандартні функтори

Інколи потрібно використати функтор який є достатньо стандартним — наприклад, для того щоб порівнювати два аргументи числового типу або використати додавання, множення і т.п. Для цього можна використати стандартний функтори, що визначений в бібліотеці <functional>

Тип	Назва	К-ть операндів	Тип, що повертається	Дія
Порівняння	equal_to	Бінарний	bool	x == y
	not_equal_to	Бінарний	bool	x!= y
	greater	Бінарний	bool	x > y
	less	Бінарний	bool	x < y
	greater_equal	Бінарний	bool	x >= y
	less_equal	Бінарний	bool	x <= y
Логичні	logical_and	Бінарний	bool	x && y
	logical_or	Бінарний	bool	x || y
	logical_not	Унарний	bool	!x
Арифметичні	plus	Бінарний	T	x + y
	minus	Бінарний	T	x - y
	multiplies	Бінарний	T	x * y
	divides	Бінарний	T	x / y
	modulus	Бінарний	T	x % y
	negate	Унарний	T	-x
Бітові (Сі++11)	bit_and	Бінарний	T	x & y
	bit_or	Бінарний	T	x | y
	bit_xor	Бінарний	T	x ^ y
	bit_not	Унарний	T	~x

Приклад:

```
#include <iostream>
#include <vector>
#include <algorithm> // mismatch, pair, copy
#include <numeric> // pair,partial_sum
#include <functional> //less, multiplies

int main(){
  std::vector<double> v1, v2(4);
  double mas1[] ={1.3,2.3,1.0,4.1};
```

```
  v1.assign(4,1.1); // v1 = 1.1,1.1,1.1,1.1
  std::copy(mas1,mas1+4,v2.begin()); // v2 = 1.3,2.3,1.0,4.1
  std::cout<<"v2=";
  for   (std::vector<double>::iterator   it   =   v2.begin();
it!=v2.end(); ++it)
     std::cout << ' ' << *it;
  std::cout <<"\n";
  std::cout<<"v1=";
  for   (std::vector<double>::iterator   it   =   v1.begin();
it!=v1.end(); ++it)
     std::cout << ' ' << *it;
  std::cout <<"\n";
  // функція mismatch з порівнянням за допомогою предікату less
  std::pair<std::vector<double>::iterator,
std::vector<double>::iterator> it;
  it   =   std::mismatch   (v1.begin(),   v1.end(),   v2.begin(),
std::less<double>() );
  std::cout << "First pair that (v1 < v2): " << *it.first;
  std::cout << " and " << *it.second << '\n';

  std::vector<double> result(4);
   // часткова сума з допомогою предікату multiplies
  std::partial_sum   (v2.begin(),   v2.end(),   result.begin(),
std::multiplies<double>());
  std::cout << "partial multiplies of v2: ";
  for (int i=0; i<4; i++) std::cout << result[i] << ", ";
  std::cout << '\n';
}
```

Результат:

```
 v2= 1.3 2.3 1 4.1
v1= 1.1 1.1 1.1 1.1
First pair that (v1 < v2): 1.1 and 1
partial multiplies of v2: 1.3, 2.99, 2.99, 12.259,
```

Література

1 Грицюк Ю.І., Рак Т.Є. Об'єктно-орієнтоване програмування мовою С++: навчальний посібник. – Львів : Вид-во Львівського ДУ БЖД, 2011. – 404 с.

2 Грицюк Ю.І., Рак Т.Є. Програмування мовою С++ : навчальний посібник. – Львів : Вид-во Львівського ДУ БЖД, 2011. – 292 с.

3 Вінник В.Ю. Алгоритмічні мови та основи програмування: мова С / В.Ю. Вінник - Житомір :ЖДТУ,2007. - 328 с.

4 Креневич А. П. С у задачах і прикладах : навчальний посібник із дисципліни "Інформатика та програмування" / А.П. Креневич, О.В. Обвінцев. – К. : Видавничо-поліграфічний центр "Київський університет", 2011. – 208 с.

5 Креневич А. П. С у задачах і прикладах : навч. посібник / А. П. Креневич, О. В. Обвінцев. – Київ : ВПЦ “Київський університет”, 2012. – 212 с.

6 В.В. Бублик, В.В. Личман, О.В. Обвінцев. Конспект лекцій з курсу "Інформатика та програмування" [Електронний ресурс] –Режим доступу до ресурсу: http://matfiz.univ.kiev.ua/informatics/lectures/

7 Бєлов Ю.А., Карнаух Т.О., Коваль Ю.В., Ставровський А.Б. Вступ до програмування мовою С++. Організація обчислень :навч. посіб. / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с.

8 Трофименко О.Г., Прокоп Ю.В., Швайко І.Г. та ін.С++. Основи програмування. Теорія та практика:Підручник / О.Г. Трофименко, Ю.В. Прокоп, І.Г. Швайко, Л.М. Буката, Л.А. Косирева, Ю.Г. Леонов, В.В. Ясинський; за ред. О.Г. Трофименко. — Одеса: Фенікс, 2010. — 544 с.

9 Електронний довідник бібліотек С++ [Електронний ресурс] – Режим доступу до ресурсу: http://www.cplusplus.com/reference/clibrary/

10 Електронний підручник з С++[Електронний ресурс] –Режим доступу до ресурсу: https://www.learncpp.com/

11 Відкритий навчальний посібник С Tutorial [Електронний ресурс] –Режим доступу до ресурсу: https://www.tutorialspoint.com/cprogramming/index.htm

12 Електронний довідник C language [Електронний ресурс] – Режим доступу до ресурсу: https://en.cppreference.com/w/c/language

13 Відкритий навчальний посібник С Programming [Електронний ресурс] –Режим доступу до ресурсу: https://www.eskimo.com/~scs/cclass

14 Вікіпедія про мову С[Електронний ресурс] –Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/C_(%D0%BC%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)

15 ANSI 89 –Programming Language C: American National Standard for Information Systems : ANSI X3.159-1989 - 338c.
16 Стандарт С99 [Електронний ресурс] –Режим доступу до ресурсу: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf
17 Чорнетка стандарту С11 [Електронний ресурс] –Режим доступу до ресурсу: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
18 CERT C Coding Standard [Електронний ресурс] –Режим доступу до ресурсу: https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard
19 Стандарт для роботи с широкими символами [Електронний ресурс] –Режим доступу до ресурсу:
20 Відкритий підручник по С++ українською мовою[Електронний ресурс] –Режим доступу до ресурсу: https://purecodecpp.com/uk/archives/920
21 Уэйт М. Язык Си / М. Уэйт, С. Прата, Д. Мартин. – М. : Мир, 1988. – 512 с
22 Керниган Б., Ритчи Д. Мова програмування Сі.: 3-е изд., пер. з англ.,- СПб .: "Невский Диалект", 2001. - 352 с .
23 Керниган, Б. Язык программирования Си. Задачи по языку Си / Б. Керниган, Д. Ритчи, А. Фьюэр. – М. : Финансы и статистика, 1985. – 279 с.
24 Дейтел Х.М. , Дейтел П.Дж. Как программировать на С. М.: Бином, 2000. -1008 с.
25 Лафоре, Роберт. Объектно-ориентированное программирование в С++. Классика Computer Science / Роберт Лафоре. – 4-е изд. : пер. с англ. – СПб. : Изд-во "Питер", 2005. – 924 с.
26 Шилдт, Герберт. Искусство программирования на С++ : пер. с англ. / Герберт Шилдт. – СПб. : Изд-во БХВ-Петербург, 2005. – 496 с.
27 Шилдт, Герберт. Полный справочник по С++ / Герберт Шилдт. – 4-е изд. : пер. с англ. – М. : Изд. дом "Вильямс", 2010. – 800 с.
28 Вакал Є.С.,Личман В.В.,Обвінцев О.В.,Бублик В.В.,Попов В.В. Задачі до курсу «Інформатика та програмування». [Електронний ресурс] –Режим доступу до ресурсу:
29 Збірник задач з дисципліни "Інформатика і програмування" Вакал Є.С., Личман В.В., Обвінцев О.В., Бублик В.В., Довгий Б.П., Попов В.В. -2-ге видання, виправлене та доповнене –К.:ВПЦ "Київський університет", 2006.–94с