Основні відмінності С та С++

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

Таблиця 6.1

Відмінності С та С++

С	С++
С підмножина (діалект) С++.	С++ надмножина (superset) С. Сі запускається з С++ але Сі не запускає С++ код.
С має 32 ключові слова .	С++ має 52 ключові слова.
С - процедурна мова	С++ процедурна але й ООП мова
Перевантаження функцій та операторів не дозволено в С.	Перезвантаження функцій та операторів дозволено в С++.
Функції в С не визначаються в структурах.	Функції визначаються структурах та класах С++.
Немає простору імен	Простори імен використовуються в С++, для запобігання колізій.
Змінні за посиланням (Reference variables) не підтримуються С.	Змінні за посиланням (Reference variables) підтримуються С++.
Віртуальних та дружніх функцій немає в С.	Віртуальні та дружні функцій є в С++.
С немає наслідування, інкапсуляції та поліморфізму (ООП)	С++ підтримує наслідування, інкапсуляції та поліморфізму (ООП)
С має лише malloc() та calloc() для алокації пам’яті та free() для деалокації.	С++ має оператор new та delete для алокації та деалокації пам’яті.
В С немає виключень	С++ підтримує виключення (Exception handling)
Форматоване введення/виведення scanf та printf для С.	Існують потоки введення/виведення та cin і cout для введення/виведення С++.
Функціональне програмування в чистому вигляді відсутнє в С.	З Стандарту С++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

Стандарт C ++ складається з трьох важливих частин:

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

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

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

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

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 Сі-код але написаний як Сі++ програма, яку можна скомпілювати, наприклад, за допомогою команди
g++ hello1.cpp.
Для того, щоб створити просту програму “Hello World” програму як Сі++ 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 Сі++

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 Сі++

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 С++ 98 використовувався варіант без використання простору імен та з додаванням “.h” в файлах бібліотек. Сучасний компілятор Сі++ **не повинен** скомпілювати такий варіант:

```
// Стандарт раніший за  C++98
#include <iostream.h>

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

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;
}
```

Введення/виведення на Сі++

На мові С++ 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 С++ 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.

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 C++98) або #include <iostream> (після C++98).
Для потокового введення даних вказується операція «>>» («читати з»). Це перевантажена операція, визначена для всіх простих типів і покажчика 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 розділяти 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). Таких помилок можна уникнути шляхом присвоювання значення керуючих символів змінним, тобто:

```
#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. Написати програму, що містить організацію виведення даних, пояснювальні повідомлення, а також символи переведення рядка.

```
/* P6_1.CPP — друк змінних */
#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"; // Нул-термінейтед рядок  name
 char middle = 'P'; // символьна змінна
 string last("Петренко"); // C++ рядок  last
 int wozrast =20; // ціла змінна
 int doplata = 2;
 float zarplata = 3009.75f; // дійсна змінна одиночної точності
 double prozent = 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 << " " << wozrast << " " << doplata << " " << zarplata << " " << prozent;
 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 << wozrast << “\t” << doplata << “\t” << zarplata << “\t” << prozent << “\n“;.
```

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 заголовочний файл <iostream>, а для файлової роботи <fstream>. (Відповідно, компілятор повинен мати доступ до відповідної об'єктної бібліотеки для правильного збору даних.)
Приклад.

```
#include <iostream>
using namespace std;
int main () {
        cout << "Привіт, світ! \ t”;
}
```

При запуску консольного додатка з'являється відкриття чотирьох потоків:
cin - для введення з клавіатурою,
cout - для буферизованого виходу на монітор,
cerr - для виведення на монітор повідомлення про помилки та забивання,
clog - буферизований аналог.
Ці чотири потоки визначені за допомогою <iostream>.
Потоки cin, cout і cerr відповідають потоковому stdin, stdout і stderr відповідно.
Щоб мати можливість використовувати стандартні потоки необхідно підключити заголовочний файл iostream або iostream.h.
Загалом різниця між стандартними заголовочними 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 заголовочні 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, за звичай 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 кожна операція помістити 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 помістити в clog може привести

до того, що вивід буде зберігатися в буфері до тих пір, поки буфер повністю не заповниться або ж поки вміст буферу не буде виведено примусово.

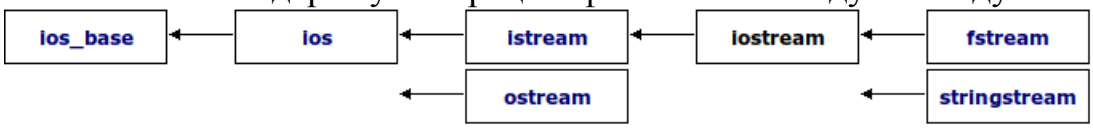
Вивід в потік виконується за допомогою операції «помістити в потік», а саме перевантаженої операції <<. Дана операція перевантажена для виводу елементів даних стандартних типів, для виводу рядків та значень вказівників. Операція << повертає посилання на об'єкт типу ostream, для якого вона викликана.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

Ієрархия класів та функції введення-виведення

Ієрархия 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 буферізованому вводе-виводі);
- **put** () // виводить символ в поток;
- **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 Сі 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 Ci++:

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 запомятовувати всі бібліотеки standard template library (STL) їх можна підключати всі одним інклюдом:

#include <bits/stdc++.h>

Тобто шаблон швидкої роботи з Ci++ 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);
   return 0;
}
```

Помітимо також, що cout << “\n”; працює швидше ніж cout << endl;. endl повільніше бо змушує обновляти поток, що незавжди 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 читаються. 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) - int 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 заголовочного файла 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 виді префікса
- 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 - ведучі 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 системис 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().

```
// P4J2.CPP — 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 заголовочний 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.

```
// P.CPP
#include <iostream>
#include <iomanip>
using namespace std;

int main ( ){
 float prod_sum; // prod_sum — сума продаж
 float nalog;
 //---------- виведення підказки
 cout << "Введить суму продаж";
 cin >> prod_sum;
```

```
 //..... обчислення податку
 nalog = prod_sum* 0.7;
 cout << "Сума: " << setprecision(2) << prod_sum;
 cout << "Податок: " << setprecision(2) << nalog << "\n";
 return 0;
}
```

Результат:

```
Введить суму продаж345.567
Сума: 3.5e+02Податок: 2.4e+02
```

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 читається 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 рідко.

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 неформатуючого вводу (тобто функцією **get**(), **getline**() або **read**()).

Розглянемо приклад, у якому використовуються дві останні функції:

```
#include <iostream>

int main(void) {
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
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 читається з вхідного потоку.

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

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 заголовочний 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 зіставляє поток з файлом. Проще говоря, 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 (перенесені 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 заголововий файл fstream, який знаходиться в просторі std.

Таблиця 6.2

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 и запису в потік, пов'язаний з файлом, здійснюється з допомогою операторів << і >>, перевантажених для занять 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 6.3

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.

```
#include <iostream>
#include <ostream>
#include <istream>
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
return 0;
}
```

Методи визначення позиції в файлі

Вказівники розташування файлів istream та ostream надають функції-члени для переміщення покажчика 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 покажчика 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.

```
#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 закрття 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 чтения 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 Сі++ 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 Сі.
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.

```
// modify boolalpha flag
#include <iostream>     // std::cout, std::boolalpha, std::noboolalpha

int main () {
  bool b = true;
  std::cout << std::boolalpha << b << '\n';
  std::cout << std::noboolalpha << b << '\n';
  return 0;
}
```

```
true
1
```

Перевантаження функцій

Перевантаження функції є властивістю С ++, 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 функцію з іншою кількістю аргументів
```
int add(int x, int y, int z){
   return x + y + z;
}
```
Примітка. На Сі такого не вдасться.
Примітка. 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 покажчиків, 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, прикріплену 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". Наступний приклад використовує посилання на int та double

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

int main () {
  //декларуємо звичайні змінні
  int    i;
  double d;

  // декларуємо змінні-посилання
  int&    r = i;
  double& s = d;

  i = 5;
  cout << "Value of i : " << i << endl;
  cout << "Value of i reference : " << r  << endl;

  d = 11.7;
  cout << "Value of d : " << d << endl;
  cout << "Value of d reference : " << s  << endl;
  return 0;
}
```

Результат роботи

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of 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 і-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 памяті 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.
C ++ 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 покажчика.
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 покажчик на початок нового виділеного блоку пам'яті.

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

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

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

Тут **<покажчик-змінна>** є покажчиком 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:

```
// Покажчик ініціалізується NULL
// Потім запитуються пам'ять для змінної
int * p1= NULL;
p1 = new int;
```

або

```
// Об’єднується декларація покажчика 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:
<покажчик-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.

<покажчик-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];
```

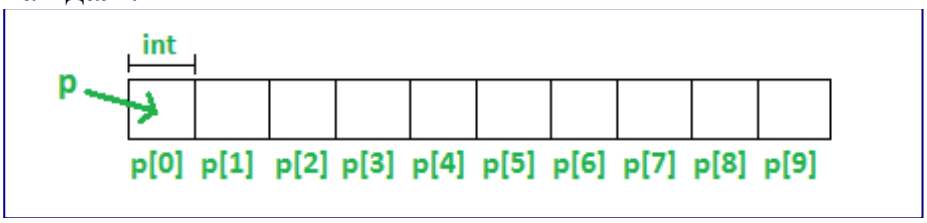
або разом з ініціалізацією

```
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 покажчик 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 NULL покажчик . Таким чином, може

бути гарною ідеєю перевірити змінні покажчика, 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 віділена память - 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 покажчика, 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. Завжди звільняйте память 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. *Видалення нульового покажчика 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:**

1) вказівник на вказівник, який містить адресу початку допоміжного масиву адрес розмір якого рівний висоті двовимірного масиву (кількості рядків);
2) допоміжний масив адрес, що зберігає адреси одновимірних масивів, які власне міститимуть дані; розмір цих масивів рівний розміру ширини двовимірного масиву (кількості стовпців);
3) 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;
        }
return 0;
}
```