Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Кафедра програмних та комп'ютерно-інтегрованих технологій

Методичні вказівки з дисципліни «Сучасні технології програмування».

(Теоретична частина)

Для студентів інституту штучного інтелекту та робототехніки

Перший (бакалаврський) рівень вищої освіти
Спеціальність: 151 – Автоматизація та комп'ютерно-інтегровані технології
Освітньо-професійна програма: Комп'ютерні технології автоматизації.

Одеса 2022

Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Кафедра програмних та комп'ютерно-інтегрованих технологій

Методичні вказівки з дисципліни «Сучасні технології програмування».

(Теоретична частина)

Для студентів інституту штучного інтелекту та робототехніки

Перший (бакалаврський) рівень вищої освіти

Спеціальність: 151 – Автоматизація та комп'ютерно-інтегровані технології

Освітньо-професійна програма: Комп'ютерні технології автоматизації.

Затверджено на засіданні

кафедри програмних та комп'ютерно-інтегрованих технологій

Протокол № 7 від 26.01.2022р.

Одеса 2022

Методичні вказівки з дисципліни Сучасні технології програмування. (Теретична частина): для студ. напряму 151 «Автоматизація та комп'ютерно-інтегровані технології» денної та заочної форм навчань./ Укл. Лисюк Г.П. - Одеса: ОП, 2022. - 97 с.

ЗМІСТ

1. ОСНОВНІ ЕТАПИ СТВОРЕННЯ ПРОГРАМИ.

Інтегроване середовище розробки (Integrated Development Environment) – це повністю самодостатнє середовище, призначене для створення, компіляції компонування та тестування програм на C++.

Редактор є інтерактивним середовищем, в якому створюється і редагується вихідний код C++.

Поряд із звичайними засобами редактор забезпечує колірне виділення різних елементів мови. Редактор автоматично розпізнає ключові конструкції мови C++ та забарвлює їх відповідно до їх значення.

Компілятор перетворює вихідний код на об'єктний код, виявляє та сповіщає про помилки в процесі компіляції. Компілятор може виявити широкий діапазон помилок, пов'язаних з некоректним або нерозпізнаним програмним кодом, а також структурні помилки.

Вихідний об'єктний код, створений компілятором, міститься в об'єктні файли. Файли з об'єктним кодом мають розширення.obj.

Компонувальник комбінує разом різні модулі, згенеровані компілятором із файлів вихідного коду, додає необхідні модулі з бібліотек і зшиває в одне ціле.

Бібліотека – це колекція попередньо написаних процедур, які підтримують і розширюють мову C++, надаючи для використання стандартні, професійно розроблені одиниці коду для виконання операцій, що часто зустрічаються.

Схема отримання модуля програми в інтегрованому середовищі показана на рис.1.1.

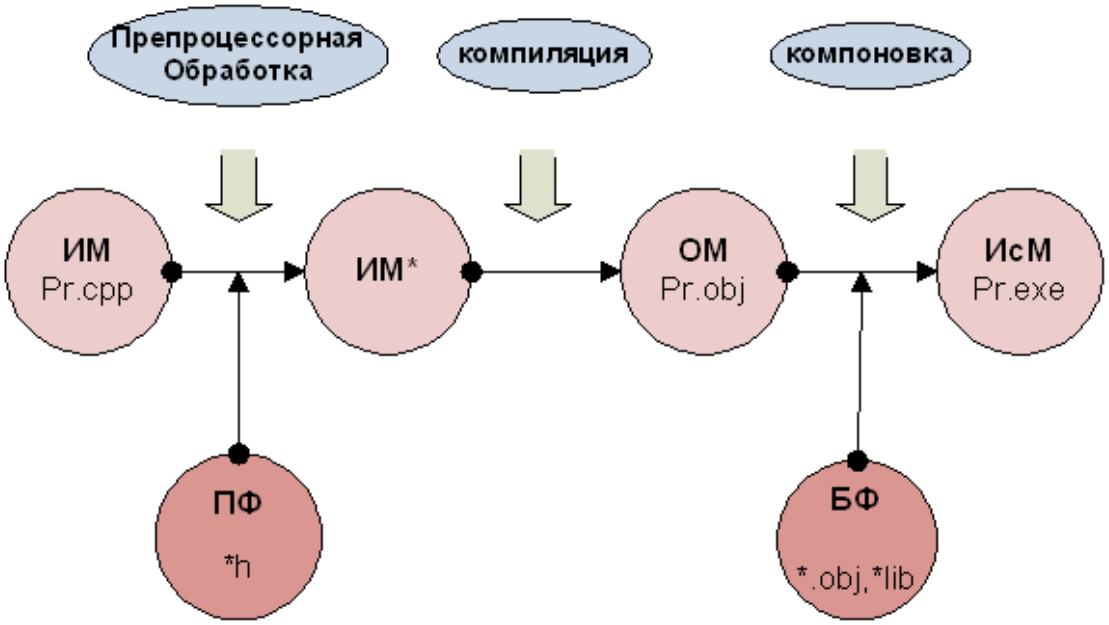


Рис.1.1. Схема розробки програмного забезпечення.

Початковий модуль програми готується за допомогою вбудованого або зовнішнього текстового редактора та розміщується у файлі з розширенням CPP.

Після цього початковий модуль програми обробляється препроцесором.

Препроцесор найкраще розглядати як окрему програму, що виконується перед компіляцією. При запуску програми препроцесор переглядає код зверху вниз, файл за файлом, у пошуку директив.

Директиви - це спеціальні команди, які починаються з символу # і НЕ закінчуються крапкою з комою. У разі необхідності, до вихідного тексту програми приєднуються файли, що підключаються (ПФ), заголовні файли (або «заголовки»), які мають розширення .h. Метою заголовних файлів є зручне зберігання набору оголошень об'єктів для їхнього подальшого використання в інших програмах.

Заголовні файли складаються із двох частин:

- Директиви препроцесора;

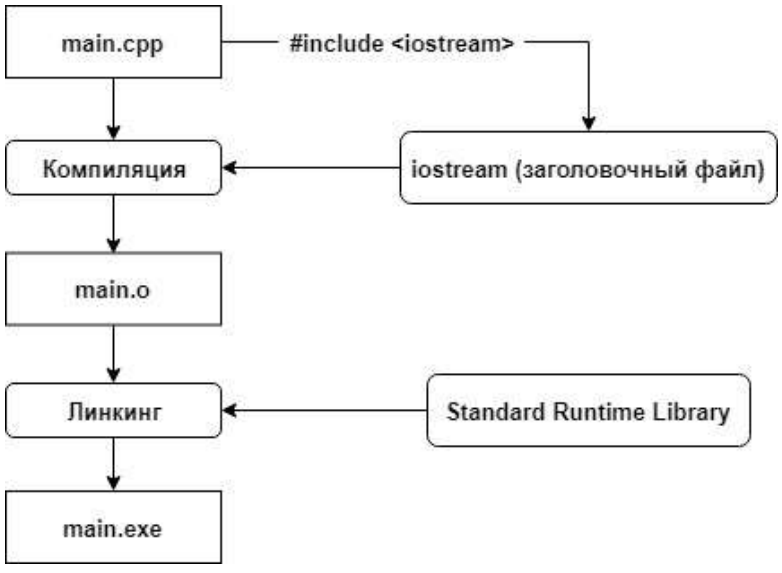
- Вміст файлу заголовка.

(У аналізованому прикладі, об'єкт cout оголошений в заголовному файлі iostream. Він визначається в Стандартній бібліотеці C++(про це говорять кутові дужки< >), яка автоматично підключається до вашого проекту на етапі лінкінгу (компонування)).

Надалі модернізований початковий модуль обробляється компілятором. Виявлені синтаксичні помилки усуваються і безпомилково відкомпільований об'єктний модуль (ОМ) поміщається у файл з розширенням obj. Потім ОМ обробляється **компонувальником**, який доповнює програму потрібними бібліотечними функціями з файлів бібліотеки (БФ). Отриманий модуль називається модулем (ІСМ) і поміщається у файл з розширенням exe , який надалі виконується.

Лінкінг— це процес зв'язування всіх об'єктних файлів, що генеруються компілятором, в єдину програму, яку виконується, яку ви зможете запустити/виконати. Це робиться за допомогою програми, яка називається лінкер (або "компонувальник").

Крім об'єктних файлів, лінкер також підключає файли зі стандартної бібліотеки C++ (або будь-якої іншої бібліотеки, яку ви використовуєте, наприклад бібліотеки графіки або звуку). Сама по собі мова C++ досить маленька і проста. Тим не менш, до нього підключається велика бібліотека додаткових функцій, які можуть використовувати ваші програми, і ці функції знаходяться у стандартній бібліотеці C++. Наприклад, якщо ви хочете вивести щось на екран, то у вас у коді має бути спеціальна команда, яка повідомить компілятор, що ви хочете використовувати функцію виведення інформації на екран із Стандартної бібліотеки C++.



Конфігурація збирання(англ. "build configuration") - це набір налаштувань проекту, які визначають принцип його побудови. Конфігурація складання складається з:

імені виконуваного файлу;

імені директорії файлу, що виконується;

імен директорій, в яких IDE шукатиме інший код та файли бібліотек;

інформації про налагодження та параметри оптимізації вашого проекту.

Інтегроване середовище розробкимає дві конфігурації збірки: «Debug» (Налагодження) та «Release» (Реліз).

Конфігурація Debug призначена для налагодження вашої програми. Ця конфігурація відключає всі оптимізації, включає інформацію про налагодження, що робить ваші програми більше і повільніше, але спрощує проведення налагодження. Режим Debug зазвичай використовується як конфігурація за замовчуванням.

Конфігурація "Release" використовується під час складання програми для її подальшого випуску. Програма оптимізується за розміром та продуктивністю і не містить додаткової інформації про налагодження.

Розглянемо найпростішу програму з однією метою – визначити, з яких компонентів, будівельних блоків вона складається.

```
//Простий приклад програми
#include <iostream>
int main()
{
int apples, oranges;                          //Оголошення двох целоч.пер
int fruit;                                    //… та ще однієї
apples = 5;                                   //Привласнення нач. значень
oranges = 6;
fruit = apples + oranges;                     //Отримати суму
std::cout << std::endl;                       //Новий рядок
std::cout << "we have" << fruit << "frut\n"; // Висновок
std::cout << std::endl;                       //Новий рядок
return 0;                                     //Вихід із програми
}
```

Ця програма містить такі елементи:

- Коментарі, що передуються двома косими рисами - //
- Директива препроцесора #include
- Заголовок функції int main()
- Тіло функції, обмежене символами {}
- Оператор , в якому для виведення повідомлень на екран використовується об'єкт *cout*
- Оператор return завершує виконання функції main()

Коментарі в мові С++.

У мові С++ коментарі позначаються подвійною похилою межею (//).

Коментар - це написана програмістом примітка, яка служить для ідентифікації розділів програми, або пояснення деяких аспектів програмного коду. Ви пишіть їх для себе чи для ваших послідовників.

Компілятор ігнорує коментарі.

Коментарім у мові С++ вважається текст від символів // і кінця рядка.

Може знаходитися в окремому рядку або в тому ж рядку, що і код програми.

У програмі С++ можна використовувати коментарі мовою С, які укладені між символами /* і */.

Оскільки коментар із мови С завершується не символом кінця рядка, а символом */ , він може тривати кілька рядків.

Коментар необхідна частина вашої програми як нагадування про те, що і навіщо ви робили.

Директива #include – файли заголовків.

Бібліотека— це набір скомпілюваного коду (наприклад, функцій), який «упаковано» для повторного використання в інших програмах. За допомогою бібліотек можна розширити можливості програмного забезпечення. Наприклад, якщо ви пишете гру, вам доведеться підключати бібліотеки звуку або графіки (якщо ви самостійно не хочете їх створювати).

Мова С++ не така вже й велика, як ви могли б подумати. Тим не менш, він іде в комплекті зі Стандартною бібліотекою С++, яка надає додатковий функціонал. Однією з часто використовуваних частин стандартної бібліотеки С++ є бібліотека iostream, яка дозволяє виводити дані на екран і обробляти введення користувача.

Після початковим коментарем у програмі перебуває директива **#include**

Вона називається директивою, оскільки вказує компілятор зробити щось, в даному випадку, вставити вміст файлу <iostream> у програму перед її компіляцією.

При підключенні заголовного файлу весь його вміст вставляється відразу після рядка #include ....

Він називається файлом заголовків, тому що зазвичай з'являється в початку програмного файлу.

Директиви препроцесора – це команди, які виконуються на фазі попередньої обробки компілятора, яка виконується перед тим, як вихідний код буде скомпілюваний в об'єктний код.

Функція main()

Навчальна програма має таку структуру:

```
int main()
{
операторы
return 0;
}
```

Ці рядки означають, що перед нами функція з ім'ям main() , і вона містить опис роботи цієї функції.

Вся сукупність наведених вище рядків становить визначення функції.

Це визначення складається із двох частин:

заголовка функції та тіла функції.

Заголовок функції у стислому вигляді визначає інтерфейс між функцією та іншою частиною програми, а тіло функції містить інструкції для комп'ютера, тобто. визначає, що робить функція.

У мові С++ кожна завершена інструкція називається **оператором**. Кожен оператор повинен закінчуватися крапкою з комою.

Заключний оператор у функції *return 0;* називається оператором повернення та завершує функцію.

Синтаксис мови С++ вимагає визначення функції main() починалося з наступного заголовка: int main().

У випадку , який буде розглянуто далі, функція С++ активізується або викликається іншою функцією, а заголовок функції описує інтерфейс між нею і тією функцією, яка її викликає.

Слово, яке стоїть перед ім'ям функції, називається повернутим типом функції, воно описує інформацію, що передається з функції назад у ту функцію, що її викликала.

Інформація в круглих дужках, що наведена за ім'ям функції, називається списком аргументів або списком параметрів. Цей список містить опис інформації , що передається з функцію, що викликає в функцію.

main() зазвичай викликається кодом початкового завантаження, який додається в програму компілятором для зв'язку програми з операційною системою.

Можно використовувати:

```
int main(void)  //функція не має жодних параметрів(перейшло з мови Сі):
void main()     //функція не повертає значення.
void main(void)
```

Тип void— це найпростіший тип даних, який означає відсутність будь-якого типу даних. Отже, змінні неможливо знайти типу void.

Оператори програми.

Кожен із операторів програми, що утворюють тіло функції main() завершується точкою з комою. Цей символ вказує на кінець оператора.

Один оператор може розповсюджуватися на кілька рядків.

Оператор програми – це базовий елемент, що визначає те, що програма робить.

Оператор - визначення дії, яке має виконати комп'ютер. Дія функції завжди виражається набором операторів, кожен із яких завершується крапкою з комою.

У наведеному прикладі у тілі функції main() представлені оператори:

- оголошення змінних,
- оператори привласнення,
- оператори виведення,
- вихід із програми.

Оператор повернення return. Коли програма завершує виконання, функція main() передає назад у операційну систему значення, що вказує результат виконання програми: успішно пройшло виконання програми чи ні.

Якщо return повертає число 0, це означає, що все добре! Ненульові значення, що повертаються, вказують на те, що щось пішло не так і виконання програми було перервано. Про return ми ще поговоримо докладніше у відповідному уроці.

Пробіл – термін, який використовується в С++ для позначення символів пробілу, табуляції, нового рядка, нової сторінки та коментарів. Пробіли служать роздільниками однієї частини оператора від іншого і дозволяють компілятору визначати, де закінчується один елемент оператора і починається інший.

Правила оформлення тексту програми, спрямовані на полегшення розуміння сенсу та підвищення наочності:

- розділяти логічні частини програми порожніми рядками
- розділяти операнди та операції пробілами
- для кожної фігурної дужки відводити окремий рядок
- у кожному рядку має бути не більше одного оператора
- обмежити довжину рядка 60-70 символами
- відступами відображати вкладеність операторів та блоків
- довгі оператори розташовувати у кількох блоках
- складати алгоритм те щоб визначення однієї функції займало, зазвичай, трохи більше одного екрана тексту
- прагнути використовувати типові заголовки фрагментів програм, типову структуру блоку та програми в цілому

2. БАЗОВІ ОПЕРАЦІЇ ВВЕДЕННЯ-ВИВОДУ. БІБЛІОТЕКА IOSTREAM.

Щоб ваш програмний продукт обробляв будь-які дані, потрібно їх описати, як ви зробили раніше, оголосивши змінні, і, нарешті, якось ввести їх у вашу програму, щоб маніпулювати ними. Природним після виконання програми буде бажання подивитися на результат. Для цього, введення та виведення інформації існують спеціальні оператори.

При підключенні заголовкового файлу iostream ми отримуємо доступ до всієї ієрархії класів бібліотеки iostream, що відповідають за функціонал введення/виведення даних (включаючи клас, який називається iostream).

Слово "stream" (тобто "потік"). По суті, введення/виведення у мові C++ реалізовано за допомогою потоків. Абстрактно, потік це послідовність символів, до яких можна отримати доступ. Згодом потік може виробляти чи споживати потенційно необмежені обсяги даних.

Ми будемо мати справу з двома типами потоків. Потік введення (або вхідний потік) використовується для зберігання даних, отриманих від джерела даних: клавіатури, файлу, мережі тощо. Наприклад, користувач може натиснути клавішу на клавіатурі, коли програма не очікує введення. Замість ігнорування натискання клавіші дані поміщаються у вхідний потік, де потім очікують відповіді від програми.

І навпаки, потік виведення (або «вихідний потік») використовується для зберігання даних, що надаються конкретному споживачеві даних: монітор, файл, принтер і т.д. При записі даних на пристрій виводу цей пристрій може бути не готовим прийняти дані негайно. Наприклад, принтер все ще може прогріватись, коли програма вже записує дані у вихідний потік. Таким чином, дані будуть знаходитися в потоці виводу, доки принтер не почне їх використовувати.

Деякі пристрої, такі як файли та мережі, можуть бути джерелами як введення, так і виведення даних.

Хороша новина полягає в тому, що програмісту не потрібно знати деталі взаємодії потоків з різними пристроями та джерелами даних, йому потрібно лише навчитися взаємодіяти з цими потоками для читання та запису даних.

Бібліотека iostream.

Бібліотека iostream надає засоби для форматованого введення-виводу.

Основу бібліотеки становлять два типи: istream, ostream.

Потік (stream) – це послідовність символів, що записується або читається з вводу-виводу.

У бібліотеці визначено чотири об'єкти водо-виводу:

cin-пов'язаний зі стандартним введенням (зазвичай це клавіатура);

cout-пов'язаний зі стандартним висновком (зазвичай це монітор);

cerr-пов'язаний зі стандартною помилкою (зазвичай це монітор), що забезпечує небуферизований висновок;

clog-пов'язаний зі стандартною помилкою (зазвичай це монітор), що забезпечує буферизований висновок.

Небуферизованный висновок зазвичай обробляється відразу, тоді як буферизованный висновок зазвичай зберігається і виводиться як блок. Оскількиclogвикористовується рідко, його зазвичай ігнорують.

Використання імен із стандартної бібліотеки.

Префікс std: означає, що наступні імена визначені всередині простору імен (namespace) на ім'я std.

::- Оператор області видимості.

Простір імен дозволяє уникнути конфліктів, причиною яких є збіг імен у певних бібліотеках.

Виведення в командний рядок

Потік виводу називаєтьсяcout, і передачі даних у нього використовується операція вставки <<. Ця операція також «вказує» напрямок руху даних. Ви вже застосовували її для виведення текстового рядка, укладеного в лапки.

Об'єкт std::cout

Об'єкт std::cout (що знаходиться в бібліотеці iostream) використовується для виведення даних на екран (у консольне вікно).

Для виведення кількох пропозицій на одному рядку оператор виводу << потрібно використовувати кілька разів, наприклад:

//Приклад програми

```
#include <iostream>
int main()
{        int a = 7;
         std::cout << "a is" << a;
         return 0;
}
```

Результат виконання програми:

a is 7

//Приклад програми

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

```
int main()
{       int num1 = 1234;        num2 = 5678;
        cout <<endl;                                    // почати новий рядок
        cout << num1< < num2;                           //вивести дві змінні
        cout << endl;                                   //завершити рядок
        return 0;                                       //завершити програму
}
```

Результат виконання програми:
12345678

Форматування виводу

Проблему, пов'язану з відсутністю прогалин між значеннями, можна виправити досить просто – вставивши пробіл у потік виведення між двома значеннями.

```
cout << num1 << num3; // Вивести два значення
```

Просто підставте замість нього оператор:

```
cout << num1 << ' ' << num2; // Вивести два значення
```

Звичайно, якщо у вас кілька рядків виводу, і ви хочете вирівняти колонки, то вам знадобляться якісь додаткові можливості, оскільки ви не знаєте скільки знаків буде в кожному значенні. З цією ситуацією можна впоратись, використовуючи те, що називається маніпуляторами.

Маніпулятор модифікує спосіб управління виведенням даних у потік (або введенням потоку).

Маніпулятори визначені в заголовку <iomanip.h>, тому вам знадобиться додати директиву #include.

Маніпулятор, який вам потрібний setw(n). Він виводить значення, що йде за ним, вирівнюючи його в полі пропусків шириною n, тобто setw(6) представить наступне за ним значення в полі шириною 6 пропусків.

```
//Приклад програми
# include <iostream>
# include <iomanip>
using namespace std;
int main()
{       int num1 = 1234; num2 = 5678;
        cout<<endl;                     // почати новий рядок
        //вивести два значення
        cout << setw(6) << num1<< setw(6) << num2;
        cout << endl;                   //завершити рядок
        return 0;                       //завершити програму
}
```

Опис результатів.

Маніпулятор setw() працює тільки з єдиним вихідним значенням, яке слідує безпосередньо за його вставкою в потік.

Ви повинні вставляти маніпулятори безпосередньо, перед кожним значенням, яке ви хочете вирівняти в межах поля певної ширини. Якщо ви вставите лише один setw(), він впливає лише перше значення, відправлене у вихідний потік за ним.

Перелік маніпуляторів:

endl - вже знайомі, переклад рядка,
ends - кінець рядка (нульовий байт),
flush - спустошити та вивести всі проміжні буфери,
dec - виводити числа у десятковому вигляді (за замовчуванням),
hex - виводити числа у шістнадцятковому вигляді,
oct - виводити числа у вісімковому вигляді,
setbase(n) - встановити систему числення для виведення чисел (n=0,2,8,10,16; 0 та 10 рівнозначні),
setw(n) - встановити ширину поля виводу,
setfill(n) - встановити символ-заповнювач до необхідної ширини поля виводу,
setprecision(n) - встановити точність виведення дробів (кількість знаків після коми).

```
//Приклад програми
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{       double x;
        cin >> x;
        cout<<setw(10)<<setfill('s')<<setprecision(3)<<x/133<<endl;
}
```

Опис результатів.

З маніпуляторами сенс такий: у полі виведення шириною 10 символів вивести результат поділу змінноїх на 133 із трьома знаками після коми. Проміжки, що залишилися, забити символами "s", потім зробити переклад рядка. Як бачите, така мішанина з маніпуляторів нормально працює

Об'єкт std::endl

Якщо текст потрібно вивести окремо (на кількох рядках), використовуйте std::endl. При використанні std::cout, std::endl вставляє символ нового рядка. Таким чином, ми переміщуємося до початку наступного рядка:

```
//Приклад програми
#include <iostream>
int main()
{std::cout << "Hi!" << std::endl;
std::cout << "My name is Anton." << std::endl;
return 0;
}
```

Результат виконання програми:

Hi!
My name is Anton.

Слово endl – це особливе значення, зване маніпулятором. При його запису в потік виводу відбувається перехід на новий рядок і скидання буфера , пов'язаного з цим пристроєм.

Керуючі послідовності

Коли ви пишіть символьний рядок, поданий у подвійні лапки, то можна включити до нього спеціальні символи, які називають керуючими послідовностями. Вони дозволяють помістити в рядок символи, які не можуть бути представлені іншим чином, завдяки тому, що вони уникають звичайного процесу інтерпретації символів.

Керуюча послідовність починається з символу зворотного слеша \ , який змушує компілятор інтерпретувати символ особливим чином.

Приклад

```
cout << endl<< “\t Output after the tabulation”;
cout << “\n\ t Output after the tabulation”;
```

Комбінація зсуває наступний за нею текст першу позицію табуляції.

\n - Керуюча послідовність символу нового рядка.

Деякі керуючі послідовності.

Керуюча послідовність	**Що робить**
\a	Видає звуковий сигнал
\n	Символ нового рядка
\'	Одиночна лапка
\\	Зворотний сліш
\b	Забій
\t	Символ табуляції
\”	Подвійна лапка
\?	Знак питання

Введення з клавіатури

Ви отримуєте введення з клавіатури через потік cin, використовуючи для цього операцію вилучення із потоку >>. Щоб прочитати два цілих значення з клавіатури, змінні num1 і num2, можна написати наступний оператор:

```
cin >> num1 >> num2;
```

Операція вилучення >> «вказує» напрям, куди передаються дані – в даному випадку,cin у кожну з двох змінних по черзі. Будь-які провідні прогалини пропускаються, і перше ціле значення, введене з клавіатури, надходить в змінну num1. Між такими значеннями повинні бути якісь пробельні символи, щоб їх можна було розділити.

Операція потокового введення завершується, коли ви натискаєте клавішу <Enter>, та виконання програми продовжується з наступного оператора.

Потокове введення та його операції автоматично розпізнають змінні та дані будь-якого фундаментального типу.

Приклад:

```
int num1 = 0, num2 = 0;
double factor = 0.0;
cin >> num1 >> factor >> num2;
```

Об'єкт std::cin

std::cin є протилежністю std::cout. Тоді як std::cout виводить дані в консоль за допомогою оператора виводу <<, std::cin отримує дані від користувача за допомогою оператора введення >>. Використовуючи std::cin ми можемо отримувати та обробляти введення користувача:

//Приклад програми

```
#include <iostream>

int main()

{       std::cout << "Enter a number: ";// просимо користувача ввести будь-яке число

        int a = 0;

        std::cin >> a;

        // отримуємо число користувача і зберігаємо його в змінну a

        std::cout << "You entered " << a << std::endl;

        return 0;

}
```

Спробуйте скомпілювати та запустити цю програму. При запуску ви побачите Enter a number:, а потім програма буде чекати, доки вкажіть число. Як тільки ви це зробите та натиснете Enter, програма виведе You entered, а потім ваше число.

Наприклад (введіть 7):

```
Enter a number: 7  You entered 7
```

3. ВИЗНАЧЕННЯ ЗМІННИХ

Навіщо пишуть програми?

Для того, щоб маніпулювати деякими даними та отримувати деякі результати.

А де зберігати ці дані та як їх знаходити, коли вони потрібні?

Для зберігання та звернення до елемента даних необхідний фрагмент пам'яті, якого можна звертатися по деякому осмисленому імені.

Кожен фрагмент пам'яті, визначений в такий спосіб, називається змінною.

Як ви вже знаєте, змінні це імена шматочків пам'яті, які можуть зберігати інформацію. Пам'ятаємо, що комп'ютери мають оперативну пам'ять, доступну програмам для використання. Коли ми визначаємо змінну, частина пам'яті відводиться їй.

Найменша одиниця пам'яті - біт (англ. "bit" від "binary digit"), який може містити або значення 0, або значення 1. Ви можете думати про біт, як про перемикач світла - або світло вимкнено (0), або включено (1). Чогось середнього між ними немає. Якщо переглянути випадковий шматочок пам'яті, то все, що ви побачите, буде ...011010100101010... або щось таке. Пам'ять організована у послідовні частини, кожна з яких має свою адресу. Подібно до того, як ми використовуємо адреси в реальному житті, щоб знайти певний будинок на

вулиці, так і тут: адреси дозволяють знайти та отримати доступ до вмісту, який знаходиться в певному місці пам'яті. Кожен біт окремо не має своєї власної адреси. Найменшою одиницею з адресою є байт (що складається з 8 бітів).

Оскільки всі дані комп'ютера - це лише послідовність бітів, ми використовуємо тип даних (або просто "тип"), щоб повідомити компілятор, як інтерпретувати вміст пам'яті. Ви вже бачили приклад типу даних: int (цілочисленний тип даних). Коли ми оголошуємо цілісну змінну, то повідомляємо компілятору, що «шматок пам'яті, який знаходиться за такою-то адресою, слід інтерпретувати як ціле число».

Коли ви вказуєте тип даних для змінної, компілятор і процесор піклуються про деталі перетворення вашого значення у відповідну послідовність біт певного типу даних. Коли ви просите ваше значення назад, воно «відновлюється» з цієї ж послідовності біт.

Пам'ять організована в блоки, які складаються з байтів, причому кожен блок має свою унікальну адресу. Одна змінна може використовувати 2, 4 або 8 послідовних адрес. Об'єм пам'яті, який використовує змінна, залежить від типу даних цієї змінної. Оскільки ми зазвичай отримуємо доступ до пам'яті через імена змінних, а не через адреси пам'яті, то компілятор може приховувати від нас всі деталі роботи зі змінними різних розмірів.

Корисно знати, скільки пам'яті займає певна змінна/тип даних.

По-перше, що більше вона займає, то більше інформації зможе зберігати. Так як кожен біт містить або 0 або 1, то 1 біт може мати 2 можливих значення.

2 біти можуть мати 4 можливі значення:

біт 0	**біт 1**
0	0
0	1
1	0
1	1

3 біти можуть мати 8 можливих значень:

біт 0	**біт 1**	**біт 2**
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0

1	0	1
1	1	0
1	1	1

Насправді, змінна з n-ним кількістю біт може мати 2n можливих значень. Оскільки байт складається з 8 біт, він може мати 28 (256) можливих значень.

Розмір змінної накладає обмеження кількості інформації, яку вона може зберігати. Отже, змінні, які використовують більше байт, можуть зберігати ширший діапазон значень.

Кожна змінна зберігає дані певного виду, і тип даних, що зберігаються, фіксується при оголошенні змінних у програмі,тобто спочатку повинні визначити, якого типу дані будуть розміщуватися в змінній, а потім заявити про своє бажання, оголосивши змінну.

Від типу змінної залежить розмір та організація фрагмента пам'яті, який ми називаємо змінною.

Конкретне значення, що зберігається змінною в кожний окремий момент часу, визначається операторами програми, і, як правило, її значення змінюється багато разів у процесі обчислень, що виконуються програмою.

Якщо у тексті програми використовується оператор

int X = 5;

То він означає, що в пам'яті виділений фрагмент для зберігання даних цілого типу, звернутися в програмі до цього фрагмента можна на ім'я X, і в даний момент в цьому фрагменті записано ціле число 5, що може змінитися.

Розмір основних типів даних C++

Розмір змінної з будь-яким типом даних залежить від компілятора та/або архітектури комп'ютера!

Мова C++ гарантує лише їх мінімальний розмір:

Категорія	**Тип**	**Мінімальний розмір**
Логічний тип даних	bool	1 байт
Символьний тип даних	char	1 байт
Цілочисельний тип даних	short	2 байти
	int	2 байти (але найчастіше 4 байти)
	long	4 байти
	long long	8 байт

Тип даних із плаваючою комою	float	4 байти
	double	8 байт
	long double	8 байт

Фактичний розмір змінних може відрізнятись на різних комп'ютерах, тому для його визначення використовують оператор **sizeof**.

Оператор sizeof - це унарний оператор, який обчислює та повертає розмір певної змінної або певного типу даних у байтах.

Ви можете скомпілювати та запустити наступну програму, щоб з'ясувати, скільки займають різні типи даних на вашому комп'ютері:

//Приклад програми

```
#include <iostream>
 int main()
{
std::cout << "bool:\t\t" << sizeof(bool) << " bytes" << std::endl;
std::cout << "char:\t\t" << sizeof(char) << " bytes" << std::endl;
std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes" << std::endl;
std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes" << std::endl;
std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes" << std::endl;
std::cout << "short:\t\t" << sizeof(short) << " bytes" << std::endl;
std::cout << "int:\t\t" << sizeof(int) << " bytes" << std::endl;
std::cout << "long:\t\t" << sizeof(long) << " bytes" << std::endl;
std::cout << "long long:\t" << sizeof(long long) << " bytes" << std::endl;
std::cout << "float:\t\t" << sizeof(float) << " bytes" << std::endl;
std::cout << "double:\t\t" << sizeof(double) << " bytes" << std::endl;
std::cout << "long double:\t" << sizeof(long double) << " bytes" << std::endl;
return 0;
}
```

Результат виконання програми:

```
bool: 1 bytes
char: 1 bytes
wchar_t: 2 bytes
char16_t: 2 bytes
char32_t: 4 bytes
short: 2 bytes
```

int: 4 bytes

long: 4 bytes

long long: 8 bytes

float: 4 bytes

double: 8 bytes

long double: 8 bytes

Зверніть увагу, оператор sizeof не використовується зтипом void, оскільки останній не має розміру.

Найменування змінних

Ім'я, яке надається змінною, називається ідентифікатором, або – ім'ям змінної.

Імена змінних можуть увімкнути

літериA - z (у верхньому або нижньому регістрі),
цифри 0-9,
підкреслення знаки.

Імена змінних повинні починатися або з літери, або з підкреслення.

Хоча імена змінних Visual C++ можуть мати довжину до 2048 символів, бажано обмежитися іменами не довшими 30 символів.

Бажано також уникати імен, що починаються з підкреслення, т.к. можливі конфлікти зі стандартними системними змінними, що мають таку саму форму.

Приклади імен:

Price
discount
value
COUNT

Символи верхнього та нижнього регістру різняться. Пробіли не можуть з'являтися в іменах.

Традиційна угода, прийнята в C++, полягає в тому, що імена, що починаються з великих літер, резервуються для найменування класів, а з великих - для назви змінних.

Ім'я змінної бажано давати осмислено, пам'ятаючи про їхнє призначення. Це допоможе уникнути багатьох помилок під час написання коду.

У C++ є зарезервовані слова, які називають ключовими словами і мають спеціальне значення всередині мови. Ці мовні конструкції під час введення коду зазвичай підсвічуються. Не слід такими словами називати свої змінні.

Приклади:

int, return, do, while…

Оголошення змінних

Оголошення змінної – оператор програми, який визначає (специфікує) ім'я змінної певного типу.

Приклад:

int value;

Цей оператор оголошує змінну на ім'я value для зберігання цілих чисел. Тип даних, який може бути збережений у змінній value, вказано ключовим словом int, тому value можна застосовувати для зберігання даних типу int, тільки цілого типу.

Оголошення змінної завжди закінчується крапкою з комою. Можливо в одному оголошенні вказати кілька імен змінних, але краще – в окремих операторах. Необхідно визначити змінну перед тим, як змінна буде задіяна у програмі.

Оголошуючи змінну, можна надати їй початкове значення.
Оголошення змінної, яке надає їй початкове значення, називається ініціалізацією.

Приклади:

```
int value = 0;
int count = 10;
int number (5);
```

Якщо не привласнити змінної початкового значення, то, як правило, вона містить довільне сміття, яке знаходилося в тій ділянці пам'яті, яка для неї виділилася. Позбавимося його - ініціалізуємо змінні при їх оголошенні.

Фундаментальні типи даних

Різновид інформації, яку може містити змінна, називається її типом даних. Усі дані та змінні у програмі повинні належати до певного типу. СтандартISO/ANSI є набір фундаментальних типів даних, визначених ключовими словами.

Фундаментальні типи даних називаються так оскільки зберігають значення типів, що становлять фундаментальні дані у комп'ютері.

Фундаментальні типи поділяються на три категорії –

- типи, що містять цілі числа,
- типи, що містять нецілочисленні значення,
- тип void, що вказує на порожню множину значень або відсутність типу.

Цілочисленні змінні

Цілочисленний тип даних - це тип, змінні якого можуть містити лише цілі числа (без дробової частини, наприклад: -2, -1, 0, 1, 2). У мові C++ є 5 основних цілих типів, доступних для використання:

Категорія	Тип	Мінімальний розмір
Символьний тип даних	char	1 байт
Цілочисельний тип даних	short	2 байти
	int	2 байти (але найчастіше 4 байти)
	long	4 байти
	long long	8 байт

Примітка: Тип char - це особливий випадок: він є цілим, так і символьним типом даних.

Основною відмінністю між цілими типами, перерахованими вище, є їхрозмір, що він більше, то більше значень зможе зберігати змінна цього типу.

Цілі численні змінні можна оголошувати з ключовим словом int.

Змінна типу int займає 4 байти пам'яті, і може зберігати як позитивні, і негативні цілі значення.

Верхній та нижній межі значень змінних типу int відповідають максимальному та мінімальному двійковому значенню зі знаком, яке може бути представлене 32 бітами. Верхня межа змінної типу int – 2 ступеня 31 +1, що дорівнює 2 147 483 647, а нижня межа - -2 31 ступеня, що відповідає -2 147 483 648.

Приклад:

```
int toeCount = 10;
```

BVisual C++ ключове слово short також визначає цілі змінні, але які займають 2 байти у пам'яті. Ключове слово short еквівалентно short int, і ви можете визначити два змінних типу short так:

```
short feetPerPerson = 2;
short int feetPerYard = 3;
```

У C++ також передбачений інший цілий тип - long, який також можна записувати як long int.

Ось як оголошуються змінні типу long:

```
long bigNumber = 10000000L;
long largeValue = 0L;
```

Цілочисленні змінні, оголошені як long, VisualC++ займають 4 байти, і можуть приймати значення від -2 147 483 648 до 2 147 483 647. Той самий діапазон можуть приймати змінні типу int.

У різних типах комп'ютерів розмір змінних може бути різним.

Загальні принципи:

Розмірність даних типу short не менше ніж 16 розрядів.

Розмірність даних типу int не менше розміру типу short.

Розмірність даних типу long не менше 32 розрядів і не менше розміру даних типу int.

Символьні типи даних

Тип даних char служить двом цілям. Він специфікує однобайтну змінну, в якій можна зберігати цілі числа в межах певного діапазону значень, або код окремого символу ASCII(American Standard code for Information Interchange)

Приклад:

```
char letter = 'A';
```

Цей код оголошує змінну на ім'я letter, ініціалізовану константою 'A'. Оскільки символ 'A' у кодуванні ASCII представлений десятковим значенням 65, ви могли б написати цей оператор так:

```
char letter = 65; // Еквівалент символу A
```

Зверніть увагу, що стандарт ISO/ANSI C++ не вимагає, щоб тип char представляв однобайтні цілі зі знаком. Це визначає вибір реалізації компілятора.

Для ініціалізації змінних char (як і інших цілих типів) ви можете використовувати шістнадцяткові константи.

Розмір, діапазон та знак типу char

У мові З++ змінних типу char завжди виділяється 1 байт. За замовчуванням char може бути як signed, так і unsigned (хоча зазвичай signed). Якщо ви використовуєте char для зберігання символів ASCII, то вам не потрібно вказувати знак змінної (оскільки signed і unsigned можуть містити значення від 0 до 127).

Але якщо ви використовуєте тип char для зберігання невеликих цілих чисел, тоді слід уточнити знак. Змінна типу char signed може зберігати числа від -128 до 127. Змінна типу char unsigned має діапазон від 0 до 255.

Модифікатори цілісних типів

Змінні цілих типів char, int, short або long зберігають за замовчуванням цілі значення зі знаком (signed), тому ви можете застосовувати їх як для збереження позитивних, так і негативних значень, оскільки для цих типів за замовчуванням прийнято модифікатор типу signed. Коли ви пишете int або long, це означає signed int або signed long відповідно.

Діапазон значень, які можуть бути збережені в змінному типу char, знаходиться в межах від -128 до +127.

Якщо ви впевнені, що вам не потрібно зберігати в змінній негативні значення (наприклад, якщо збираєтеся записувати в неї кількість миль, які ви проїжджаєте за кермом на тиждень), то ви можете специфікувати змінну як unsigned:

unsigned long mileage = OUL;

Булівський тип

Логічні змінні— це змінні, діапазон яких складається лише з двох можливих значень: true (1) та false (0).

Для оголошення логічної змінної використається ключове слово bool.

Булівські змінні – це такі змінні, які можуть зберігати лише два значення: true чи false.

Ім'я змінної типу bool оголошується так:

```
bool testRestult;
bool colorIsRed=true;
```

Насправді логічні значення не зберігаються як true або false. Вони обробляються як цілих чисел: замість true — одиниця, замість false — нуль.

Отже, якщо ми спробуємо вивести логічні значення за допомогою std::cout, То побачимо або 0, або 1:

//Приклад програми

```
#include <iostream>
 int main()
{       std::cout << true << std::endl; // замість true одиниця
        std::cout << !true << std::endl; // замість !true нуль
        bool b(false);
        std::cout << b << std::endl; // b - false (0)
        std::cout << !b << std::endl; // !b - true (1)
        return 0;
}
```

Результат виконання програми:

```
1001
```

Типи з плаваючою точкою

Числові змінні, що не належать до цілих, зберігаються як числа з плаваючою точкою. Число з плаваючою точкою може бути виражене у вигляді десяткового значення на кшталт 112,5 або в експоненційному вигляді, такому як 1,125E2, де десяткова частина множиться на 10 ступеня, зазначеної після E

(Експонента). Таким чином, останнє число – це 1,125x10 у ступеню 2, що дорівнює 112,5.

Константи з плаваючою точкою повинні включати десяткову точку або експоненту, або те й інше. Якщо ви записуєте числове значення без них, отримуєте ціле.

Є три типи даних з плаваючою точкою: float, double та long double. Мова C++ визначає лише їх мінімальний розмір (як і з цілими типами). Типи даних з плаваючою точкою завжди є signed (тобто можуть зберігати як позитивні, і негативні числа).

Категорія	**Тип**	**Мінімальний розмір**	**Типовий розмір**
Тип даних із плаваючою точкою	float	4 байти	4 байти
	double	8 байт	8 байт
	long double	8 байт	8, 12 або 16 байт

Змінна типу double займає 8 байт пам'яті та зберігає значення, точність яких визначається приблизно 15 десятковими знаками. Діапазон їх значень набагато ширший, ніж можна виразити 15-ма знаками – починаючи від 1,7x10 у ступені -308 та закінчуючи 1,7x10 у ступені 308, позитивні та негативні.

Якщо вам не потрібна 15-значна точність і не потрібні дуже великі діапазони значень, які забезпечує тип double, можете скористатися ключовим словом float для оголошення змінних з плаваючою точкою, що займають 4 байти.

Приклад:

```
float pi=3.14159f;
```

Цей оператор визначає змінну рі з початковим значенням 313159. Символ f в кінці константи показує, що вона має тип float. Без f константа мала би тип double. Змінні, оголошені з типом float, мають точність приблизно 7 десяткових знаків і допускають значення від 3,4x10 у ступені -38 до 3,4x10 у ступені 38, позитивні та негативні.

Якщо потрібно використовувати ціле число зі змінною типу з плаваючою точкою, тоді після цього числа потрібно поставити роздільну точку і нуль. Це дозволяє відрізняти змінні цілих типів від змінних типів з плаваючою комою.

```
double n=5.0;
```

Фундаментальні типи ISO/ANSI C++

У таблиці представлений список всіх фундаментальних типів ISO/ANSI C++, а також діапазони їх допустимих значень Visual C++

ТИП	РОЗМІР У БАЙТАХ	ДІАПАЗОН ЗНАЧЕНЬ
bool	1	true або false
char	1	від -128 до +127
signed char	1	від -128 до +127
unsigned char	1	від 0 до 255

short	2	від -32768 до +32767
unsigned short	2	від 0 до 65535
int	4	від -2147483648 до 2147483647
unsigned int	4	від 0 до 4294967296
long	4	від -2147483648 до 2147483647
unsigned long	4	від 0 до 4294967296
float	4	+-3,4x10+-38приблизно з 7-значною точністю
double	8	+-1,7x10+-308приблизно з 15-значною точністю

4. ТИПИ ЗМІННИХ ТА ПРИВЕДЕННЯ.

Обробка арифметичних виразів

Щоб правильно обчислювати вирази (наприклад, 4+2*3), ми повинні знати, що роблять певні оператори та в якому порядку вони виконуються. Послідовність, де вони виконуються, називається пріоритетом операцій. Наслідуючи звичайні правила математики (у якій множення слід перед додаванням), вираз, наведений вище, обробляється наступним чином: 4 + (2 * 3) = 10.

У мові C++ всі оператори (операції) мають рівень пріоритету. Ті, у яких він вищий, виконуються першими. Пріоритет операцій множення та розподілу вищий, ніж в операціях складання та віднімання. Компілятор використовує пріоритет операторів для визначення порядку обробки виразів.

А що робити, якщо у двох операторів у вираженні однаковий рівень пріоритету і вони розміщені поруч? Яку операцію компілятор виконає першою? А тут уже компілятор використовуватиме правила асоціативності, які вказують напрямок виконання операцій: зліва направо або праворуч наліво. Наприклад, у вираженні 3*4/2 операції множення та поділу мають однаковий рівень пріоритету (5-й рівень). А асоціативність п'ятого рівня відповідає виконанню операцій зліва направо, таким чином: (3*4)/2=6.

Обчислення C++ можуть виконуватися тільки між однотипними значеннями. Коли вираз включає змінні, або константи різних типів, для кожної операції компілятор повинен виконувати перетворення типу одного операнда до типу іншого.

Процес перетворення типів називається приведенням.

Якщо операнди різних типів даних, компілятор обчислює операнд з найвищим пріоритетом і неявно конвертує тип іншого операнда в такий же тип, як у першого.

Наприклад, якщо значення типу double складається зі значенням цілого типу, то ціле значення спочатку перетворюється в double , після чого виконується додавання. При цьому змінна, що містить вихідне значення, не змінюється. Компілятор зберігає перетворене значення у тимчасовій пам'яті, яка втрачається після завершення обчислень.

Пріоритет типів операндів:

long double (найвищий);

double;

float;

unsigned long long;

long long;

unsigned long;

long;

unsigned int;

int (найнижчий).

Правила приведення операндів.

- Якщо кожен із операндів має тип long double, то другий перетворюється на long double.
- Якщо будь-який з операндів має тип double, другий перетворюється на тип double.
- Якщо будь-який з операндів має тип float, то другий перетворюється на тип float.
- Будь-який операнд типу char, signed char, unsigned char, short або unsigned short перетворюється на тип int.
- Перерахований тип перетворюється спочатку в int, unsigned int, long або unsigned long, залежно від того, якого з них достатньо, щоб вмістити діапазон перелічників.
- Якщо один з операндів типу unsigned long, то інший перетворюється на unsigned long.
- Якщо один із операторів типу long, а інший типу unsigned int, то обидва перетворюються на unsigned long.
- Якщо будь-який з операндів типу long, то другий перетворюється на тип long.

Базовий принцип простий: завжди перетворюється значення типу, що має більш обмежений діапазон, до типу другого значення.

Приклад:

Нехай оголошено послідовність змінних

```
double value = 31.0;
int count = 16;
float many = 2.0f;
charnum = 4;
```

Можна припустити, які компілятор виконає у довільному арифметичному вираженні.

```
value = (value - count) * (count - num) / many + num / many;
```

Перша операція - (value - count) Застосовно правило 2. count перетворюється на double. В результаті виходить значення типу 15.0 double.

Друга операція - (count - num) Застосовується правило 4. num перетворюється з char в int.

В результаті виходить 12 типів int.

Третя операція – перемноження двох перших результатів - 15.0 типу double та 12 типу int. Застосовується правило 2. І 12 перетворюється на 12.0 з типом double.

В результаті виходить значення 180.0 із типом double.

4-а операція – отриманий результат має бути поділений на багато типу float.

Застосовується правило 2. багато перетворюється на double.

В результаті виходить 90.0 з типом double.

5-а операція - num/many. Застосовується правило 3. num перетворюється з char у float. В результаті виходить значення типу 2.0f float.

6-а операція – до double значення 90.0 додається float значення 2.0 f. Застосовується правило 2, яке вимагає перетворення 2.0 f на double значення 2.0 I остаточний результат 92.0 присвоюється value.

Приведення в операторах присвоєння.

Праворуч від оператора привласнення може бути записано вираз, тип якого відрізнятиметься від типу змінної, що знаходиться ліворуч від нього. Це може змінити значення та інформація буде втрачена.

Приклад

```
int number = 0;
float decimal = 2.5f;
number = decimal;
```

Значення number дорівнює 2.

Явне приведення

Коли змішані вирази включають базові типи, можна примусово вказати приведення одного типу до іншого, використовуючи явні приведення:

static_cast < тип якого привести> (вираз)

Ключове слово **static_cast** відбиває те що, що приведення виконується статично – тобто. коли програма компілюється.

Ефект операціїstatic_cast полягає в перетворенні значення результату обчислення виразу типу, який вказаний між кутовими дужками. Вираз може бути будь-яким – від окремої змінної, до складної складової.

Приклад

```
double value1 = 10.5;
double value2 = 15.5;
int whole_number = static_cast<int>(value1) + static_cast<int>(value2);
```

Ініціалізуючим виразом для whole_number є сума цілих частин value1 та value2, в яких залишається 10.5 та 15.5 відповідно, значення 10 та 15, породжені привидом, зберігаються лише тимчасово, для використання у обчисленні суми, а потім губляться.

5. ОПЕРАТОРИ. УНАРНІ ОПЕРАТОРИ. БІНАРНІ АРИФМЕТИЧНІ ОПЕРАТОРИ. ТЕРНАРНЫЕ.

Вираз- Це математичний об'єкт, який має певне значення. Точніше: вираз - це комбінація літералів, змінних, функцій та операторів, яка генерує (створює) певне значення.

Літерал— це фіксоване значення, яке записується безпосередньо у вихідному коді (наприклад, 7 або 3.14159).

Літерали, змінні та функції ще відомі як операнди.

Операнди- це дані, з якими працює вираз. Літерали мають фіксовані значення, змінним можна надавати значення, функції ж генерують певні значення (залежно від **типу повернення**, Винятком є функції типу void).

За допомогою операторів ми можемо поєднати операнди для отримання нового значення. Наприклад, у виразі 5+2, + є оператором. За допомогою+ми об'єднали операнди 5 і 2 для отримання нового значення (7).

Ви, мабуть, вже добре знайомі зі стандартними арифметичними операторами зі шкільної математики: додавання (+), віднімання (-), множення (*) та розподіл (/). Знак рівності = є оператором присвоєння. Деякі оператори складаються з більш ніж одного символу, наприклад, оператор рівності = = що дозволяє порівнювати між собою два певні значення.

Примітка:Дуже часто новачки плутають оператор присвоєння (=) з оператором рівності (= =). За допомогою оператора привласнення (=) ми привласнюємо змінною певне значення. За допомогою оператора рівності (= =) ми перевіряємо, чи рівні між собою два певні операнди.

Оператори бувають трьох типів:

Унарні. Працюють із одним операндом. Наприклад, оператор − (мінус). У виразі −7, оператор−застосовується тільки до одного операнда (7), щоб створити нове значення (−7).

Бінарні. Працюють із двома операндами (лівим та правим). Наприклад, оператор +. У виразі 5+2, оператор+працює з лівим операндом (5) та правим (2), щоб створити нове значення (7).

Тернарні. Працюють із трьома операндами (у мові C++ є лише одна **тернарний оператор**).

Зверніть увагу, деякі оператори можуть мати кілька значень. Наприклад, оператор − (мінус) може використовуватися у двох контекстах: як унарний для зміни знака числа (наприклад, конвертувати 7 в –7 і навпаки), і як бінарний для виконання арифметичної операції віднімання (наприклад, 4 - 3).

Бінарні арифметичні оператори.

Існує 5 бінарних операторів.

Оператор	**Символ**	**Приклад**	**Операція**
Додавання	+	x + y	x плюс y
Віднімання	−	x − y	x мінус y
Розмноження	*	x * y	x помножити на y
Поділ	/	x/y	x розділити на y
Поділ із залишком	%	x % y	Залишок від поділу x на y

Оператори складання, віднімання та множення працюють так само, як і у звичайній математиці. Поділ та поділ із залишком розглянемо детально.

Розподіл цілих чисел і чисел типу з плаваючою точкою.

Оператор поділу має два режими. Якщо обидва операнди є цілими числами, то оператор виконує цілий поділ. Тобто. будь-який дріб (більше/менше) відкидається і повертається ціле значення без залишку, наприклад, 7 / 4 = 1.

Якщо один або обидва операндитипу з плаваючою точкою, тоді буде виконуватися розподіл типу з плаваючою точкою. Тут вже дріб присутній. Наприклад, вирази 7.0/3 = 2.333, 7/3.0 = 2.333 або 7.0/3.0 = 2.333 мають один і той же результат.

Спроби поділу на 0 (або 0.0) стануть причиною збою у вашій програмі, і це правило не слід забувати!

Поділ із залишком (%) працює тільки з цілими операндами і повертає залишок від цілісного поділу.

Наприклад:

Приклад №1: 7/4 = 1 із залишком 3, таким чином, 7 % 4 = 3.

Приклад №2: 25/7 = 3 із залишком 4, таким чином, 25 % 7 = 4. Залишок становить не дріб, а ціле число.

Приклад №3: 36 % 5 = 7 із залишком 1. У числі 36 лише 35 ділиться на 5 без залишку, тому 36 − 35 = 1, 1 — це залишок та результат.

Цей оператор найчастіше використовують для перевірки поділу без залишку одних чисел на інші. Якщо x % y == 0, то x ділиться на y без залишку.

Якщо один із операндів оператора поділу із залишком є негативним, то результат a % b був того ж знака, що і значенняа.

Інкремент та декремент.

Операції інкременту (збільшення на 1) та декременту (зменшення на 1) змінних настільки використовуються, що вони мають власні оператори в мові C++. Крім того, кожен з цих операторів має дві версії застосування: префікс та постфікс.

Оператор	**Символ**	**Приклад**	**Операція**
Префіксний інкремент(преінкремент)	++	++x	Інкремент x, потім обчислення x
Префіксний декремент (пре-декремент)	––	––x	Декремент x, потім обчислення x
Постфіксний інкремент (пост-інкремент)	++	x++	Обчислення x, потім інкремент x
Постфіксний декремент (пост-декремент)	––	x––	Обчислення x, потім декремент x

З операторами інкременту/декременту версії префікс все просто. Значення змінної x спочатку збільшується/зменшується, а потім обчислюється.

Наприклад:

```
int x = 5;
```

```
int y = ++x;   //  x = 6 та 6 присвоюється змінною y
```

А ось з операторами інкременту/декременту версії постфікс дещо складніший. Компілятор створює тимчасову копію змінної x, збільшує або зменшує оригінальний x (не копію), а потім повертає копію. Тільки після повернення копія x видаляється.

Наприклад:

```
int x = 5;
int y=x++;   //  x = 6, але змінної y надається 5
```

Розглянемо код вище, детально. По-перше, компілятор створює тимчасову копію x, яка має те саме значення, що й оригінал (5). Потім збільшується початковий x з 5 до 6. Після цього компілятор повертає тимчасову копію, значенням якої є 5, і привласнює її змінною y. Тільки після цього копія x знищується. Отже, у наведеному вище прикладі, ми отримаємо y = 5 і x = 6.

Розглянемо ще один приклад, що показує різницю між версіями префікс та постфікс:

//Приклад програми

```
1
        #include <iostream>
2
3
        int main()
4
        {
5
                int x = 5, y = 5;
6
                std::cout << x << " " << y << std::endl;
7
                std::cout << ++x << " " << --y << std::endl; //версія префікс
8
                std::cout << x << " " << y << std::endl;
9
                std::cout << x++ << " " << y-- << std::endl; //версія постфікс
10
                std::cout << x << " " << y << std::endl;
11
                return 0;
12
        }
```

Результат виконання програми:

```
5 56 46 46 47 3
```

У рядку №7 змінні x і y збільшуються/зменшуються на одиницю безпосередньо перед обробкою компілятором, тому відразу виводяться їх нові значення. А в рядку №9 тимчасові копії (x = 6 та y = 4) відправляються в cout, а після цього вихідні x і y збільшуються/зменшуються на одиницю. Саме тому зміни значень змінних після виконання операторів версії постфіксу не видно до наступного рядка.

Префікс збільшує/зменшує значення змінних перед обробкою компілятором, постфікс – після обробки компілятором.

Порядок виконання операцій.

Порядок виконання операцій упорядковує операції у порядку пріоритетів. У будь-якому вираженні операції з вищим пріоритетом завжди виконуються першими, за ними виконуються операції з наступним за зростанням пріоритетом тощо, аж до тих, чий пріоритет найнижчий.

Порядок виконання операцій C++

Операції	Асоціативність
::	ліва
() [] -> .	ліва
!- +(унарна) –(унарна) ++ -- &(унарна) *–(унарна)	права
.*(унарна) ->*	ліва
*/%	ліва
+ -	ліва
<< >>	ліва
< <= > >=	ліва
== !=	ліва
&	ліва
^	ліва
|	ліва
&&	ліва
||	ліва
?: (умовна операція)	права
= *= /= %= += -= &= ^= |= <<= >>=	права
'	ліва

Операції з найвищим пріоритетом знаходяться у верхній частині таблиці. Всі операції, які вказані в одному осередку таблиці, мають однаковий пріоритет, якщо у виразі немає дужок, операції з рівним пріоритетом виконуються в послідовності, що визначається їхньою асоціативністю.

Якщо асоціативність ліва, то ліва операція виконується першою, потім послідовно виконуються операції всього виразу - зліва направо.

Вираз a+ b + c + d виконується, як записано (((a+ b)+ c) + d), оскільки бінарна операція має ліву асоціативність.

Завжди можна змінити пріоритети операцій у виразі за допомогою дужок. Використання дужок полегшує читання коду, рекомендується їх використання для впевненості та зручності.

Умовний тернарний оператор.

Умовний (тернарний) оператор (позначається як ?:) є єдиним тернарним оператором у мові C++, який працює з трьома операндами. Через це його часто називають просто "тернарний оператор".

Оператор? надає скорочений спосіб (альтернативу) розгалуження if/else.

if/else:

```
if (умова)
    вираз;
else
    інше_вираз;
```

Можна записати як:

(Умова) ? вираз: інше_вираз;

Зверніть увагу, що операнди умовного оператора повинні бути виразами.

Порада: завжди укладайте в дужки умовну частину тернарного оператора, а найкраще весь тернарний оператор.

6. ЧАС ЗБЕРІГАННЯ ТА ОБЛАСТЬ ВИДИМОСТІ.

Усі змінні мають обмежений час життя під час виконання програми. Вони з'являються у точці, де ви їх оголосили. а потім у деякій точці вони зникають – пізніше моменту завершення програми. Наскільки довго існуєконкретна змінна, визначається властивістю, що називається часом зберігання. Існують три різні види часу зберігання змінних.

- автоматичне
- статичне
- динамічний

Яка з них матиме змінна - залежить від того, як ви її створюєте. Інша властивість, властиве змінним - це область видимості.

Область видимості змінної – це частина програми, протягом якої ім'я цієї змінної визначено.

Поза цією областю видимості ви не можете посилатися на її ім'я – будь-яка спроба зробити це викликає помилку компіляції.

Час життя та область видимості – різні речі.

Час життя – це період під час виконання програми, починаючи з моменту першого оголошення змінної до моменту її знищення та звільнення зайнятої нею пам'яті іншого використання.

Область видимості змінної – це частина програмного коду, де змінна доступна.

Автоматичні змінні

Змінні, оголошені всередині блоку, називаються автоматичними (локальними), і такі говорять, що вони локальна область видимості, чи область видимості блоку. Автоматична змінна видима з точки, в якій вона оголошена, і до кінця блоку, що містить її оголошення.

(Блоки(або «складові оператори») - це група операторів, які обробляють компілятором як одна інструкція. Блок починається з символу { і закінчується символом}. Наприкінці складеного оператора точка з комою не ставиться.)

Простір, який займає автоматична змінна, виділяється автоматично в області пам'яті, яка називається стеком, яка спеціально призначається для цієї мети.

Наприклад, розглянемо таку програму:

```
#include <iostream>

int main()

{       int x(4);       // змінна x створюється та ініціалізується тут
```

```
        double y(5.0); // змінна y створюється та ініціалізується тут
        return 0;
}       // x і y виходять із області видимості і знищуються тут
```

Оскільки змінні x і y визначені всередині блоку, який є головною функцією, вони обидві знищуються, коли main() завершує своє виконання.

Змінні, визначені всередині вкладених блоків, знищуються, як тільки закінчується вкладений блок:

```
#include <iostream>
int main()              // зовнішній блок
{       int m(4);       // змінна m створюється та ініціалізується тут
        {               // Початок вкладеного блоку
                double k(5.0);  // змінна k створюється та ініціалізується тут
        }       // k виходить з області видимості і знищується тут
// Змінна k може бути використана тут, оскільки вона вже знищена!
return 0;
}       // змінна m виходить з області видимості і знищується тут
```

Такі змінні можна використовувати лише усередині блоків, у яких визначено. Оскільки кожна функція має свій власний блок, змінні з однієї функції ніяк не стикаються і не впливають на змінні з іншої функції:

```
#include <iostream>
void someFunction()
{       int value(5);   // value визначається тут
        // value можна використовувати тут
}       // value виходить з області видимості і знищується тут
int main()
{
// value не можна використовувати всередині цієї функції
someFunction();
// value тут також не можна використовувати
return 0;
}
```

У різних функціях можуть бути змінні або параметри із однаковими іменами. Це добре, тому що не потрібно турбуватися про можливість виникненняконфліктів іменміж двома незалежними функціями. У прикладі нижче в обох функціях є змінні x та y. Вони навіть не підозрюють про існування один одного:

```
#include <iostream>
 //Параметр x можна використовувати лише всередині функції add()
 int add(int x, int y)
// параметр x функції add() створюється тут
{
        return x + y;
} // параметр x функції add() знищується тут
 // Змінну x функції main() можна використовувати лише всередині функції main()
int main()
{
        int x = 5; // змінна функції x main() створюється тут
        int y = 6;
// значення x функції main() копіюється в змінну x функції add()
std::cout << add(x, y) << std::endl; return 0;
} // змінна x функції main() знищується тут
```

Вкладені блоки вважаються частиною зовнішнього блоку, де вони визначені. Отже, змінні, визначені у зовнішньому блоці, можуть бути помітні і всередині вкладеного блоку:

```
#include <iostream>
int main()
{ // початок зовнішнього блоку
 int x(5);
        { // Початок вкладеного блоку
                int y(7);
                // Ми можемо використовувати x та y тут
                std::cout << x << " + " << y << " = " << x + y;
        } // змінна y знищується тут
 // Змінну y тут не можна використовувати, оскільки вона вже знищена!
 return 0;
} // змінна x знищується тут
```

Розміщення оголошень змінних

Найважливіший аспект. який слід враховувати при оголошенні змінної – це яка має бути область видимості змінної. Крім цього, зазвичай слід розміщувати оголошення змінної ближче до того місця, де вона буде вперше використана у програмі.

Глобальні змінні

Змінні, які оголошені поза блоками та класами, називаються глобальними і вони мають глобальну область видимості. Це означає, що вони доступні всім функціям файлу, починаючи з точки, де вони були оголошені.

Якщо їх оголосити на початку програми, вони будуть доступні в будь-якому місці файлу.

Глобальні змінні також за умовчанням мають статичний час життя. Глобальні змінні зі статичним часом життя існують з початку виконання програми досі завершення. Якщо не визначити початкове значення глобальної змінної. то за умовчанням вона ініціалізується банкрутом.

Статичні змінні

Часто виникає необхідність оголосити змінну, що має область видимість у межах блоку, але забезпечити статичний час зберігання. Специфікатор static забезпечує таку можливість.

Статична змінна існує протягом усього життя програми, навіть якщо вона оголошена всередині блоку і доступна тільки в ньому.

Тобто вона створюється (і ініціалізується) лише один раз, а потім зберігається протягом виконання всієї програми.

Оголошується статична змінна так:

```
static int count;
```

Якщо не надати статичну змінну початкове значення при оголошенні, вона буде ініціалізована нулем.

Розглянемо різницю між змінними з автоматичною та статичною тривалістю життя.

Автоматична тривалість життя (за умовчанням):

```
#include <iostream>
 incrementAndPrint()
{       int value = 1;  // автоматична тривалість життя (за умовчанням)
        ++value;
        std::cout << value << std::endl;
} // змінна value знищується тут
 int main()
{       incrementAndPrint();
        incrementAndPrint();
        incrementAndPrint();
}
```

Щоразу, при виклику функції incrementAndPrint(), створюється змінна value, якій присвоюється значення 1. Функція incrementAndPrint() збільшує значення змінної до 2, а потім виводить його. Коли incrementAndPrint() завершує виконання, змінна виходить з області видимості і знищується.

Отже, результат виконання програми: 222

Тепер розглянемо статичну версію. Єдина різниця між цими двома програмами лише у додаванні ключового слова static до змінної.

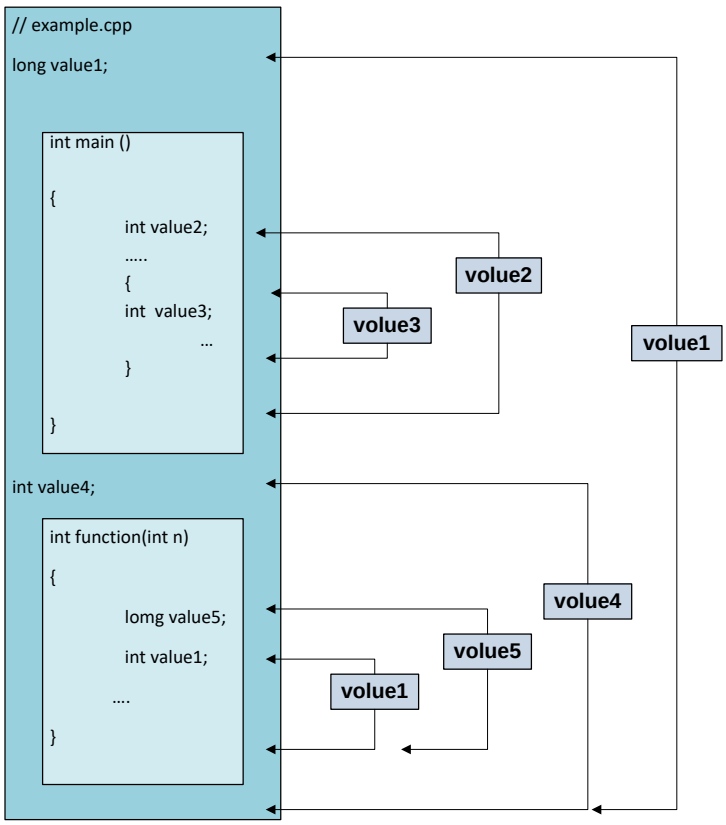
Статична тривалість життя:

```
#include <iostream>
 incrementAndPrint()
{       static int s_value = 1; // змінна s_value є статичною
        ++s_value;
        std::cout << s_value << std::endl;
} // змінна s_value не знищується тут, але стає недоступною
 int main()
{       incrementAndPrint();
        incrementAndPrint();
        incrementAndPrint();
}
```

Оскільки змінна s_value оголошена статичною (за допомогою ключового слова static), вона створюється і ініціалізується лише один раз. Крім того, виходячи з області видимості, вона не знищується. Щоразу, при виклику функції incrementAndPrint(), значення s_value збільшується.

Результат виконання програми: 234

Область видимости переменной



Змінна value1, яка є на початку файлу, оголошена з глобальною областю видимості. як і value4, яка виникає після функції main(). Область видимості кожної глобальної змінної визначається точки її визначення остаточно файла. Навіть незважаючи на те, що value4 існує в момент початку виконання програми, до неї не можна звернутися з тіла main(), тому що main() не знаходиться у її області видимості. Щоб main() могла звернутися до змінної value4, її оголошення слід перемістити на початок файла.

Обидві змінні – і value1, і value4 – за замовчуванням будуть ініціалізовані нулем, що відрізняє їх від автоматичних змінних. Локальна змінна value1 у function () приховує глобальну змінну з тим самим ім'ям.

Разом:

Область видимості ідентифікатор визначає, де він доступний для використання. До ідентифікатора, який знаходиться поза зоною видимості, доступ закрито.

Змінні з **локальною/блоковою областю видимості** доступні лише у межах блоку, де вони оголошені. Це:

- локальні змінні;
- параметри функції.

Змінні з **глобальною/файловою областю видимості** доступні будь-де файлу. Це: глобальні змінні.

Тривалість життя змінної визначає, де створюється і де знищується.

Змінні з **автоматичною тривалістю життя** створюються у точці визначення та знищуються при виході з блоку, у якому визначено. Це: звичайні локальні змінні.

Змінні зі **статичною тривалістю життя** створюються, коли програма запускається, і знищуються за її завершенні. Це:

- глобальні змінні;
- статичні локальні змінні.

Змінні з **динамічною тривалістю життя** створюються та знищуються на запит програміста. Це: динамічні змінні

7. ПОРІВНЯННЯ ЗНАЧЕНЬ.

Завдання прийняття рішень потребує механізму порівняння сутностей. Для цього потрібні операції відносин. Оскільки вся інформація представлена у числовому вигляді, порівняння числових значень – суть всього механізму прийняття рішень.

Існує шість фундаментальних операцій для порівняння двох доступних значень:

< менше ніж
> більшечим
= = одно
<= менше чи одно
>= більше чи одно
!= не одно

Кожна з цих операцій порівнює значення двох своїх операндів та повертає одне з двох можливих значень типу bool:

t r u e - якщо порівняння істинне,
f a l s e - якщо ні.

Можна побачити, як це працює, розглянувши кілька простих прикладів порівнянь.

//Приклади використання цих операторів на практиці:

```
#include <iostream>
int main()
{
std::cout << "Enter an integer: ";
int x;
std::cin >> x;

std::cout << "Enter another integer: ";
int y;
std::cin >> y;

if (x == y)
std::cout << x << " equals " << y << "\n";
if (x! = y)
std::cout << x << " does not equal " << y << "\n";
if (x > y)
std::cout << x << " is greater than " << y << "\n";
if (x < y)
std::cout << x << " is less than " << y << "\n";
if (x >= y)
std::cout << x << " is greater than or equal to " << y << "\n";
if (x <= y)
std::cout << x << " is less than or equal to " << y << "\n";

return 0;
}
```

Результат виконання програми:

Enter an integer: 4Enter another integer: 54 does not equal 54 is less than 54 is less than or equal to 5

Приклади.

Припустимо, що оголошені цілочисленні змінні i та j зі значеннями 10 та -5 відповідно.

Усі подані нижче вирази повертають значення true:

i > j i! = j j > -8 i<= j + 15

Припустимо, що ми визначили такі змінні:

char first = 'A', last = 'Z';

first == 65 first < last 'E' <= first first != last

Усі вирази порівнюють значення кодів ASCII.

Перший вираз повертаєtrue, т.к.first ініціалізована символом 'A', що еквівалентно десятковому числу 65.

Другий вираз перевіряє, чи менше значенняf i r s t , яке дорівнює 'A', ніж значення last, яке дорівнює 'Z'. Якщо заглянути в таблицю кодів символів ASCII , ми побачимо, що великі літери представлені послідовними числовими величинами - від 65 до 90, причому 65 представляє 'A', а 90-'Z', тому друге порівняння також поверне true. Третій вираз поверне false, тому що 'E' більше, ніж значення first.

Останнє вираз поверне true, оскільки 'A' безумовно не дорівнює 'Z'.

Тепер розглянемодещо складніші порівняння чисел. Маючи змінні, визначені таким чином:

Int i = -10, j = 20;

double x = 1.5, y = -0.25E-10;

погляньте на такі вирази:

-1 < y

j < (10 - i)

2. 0 * x >= (3 + y)

Як бачите, як операнди порівняняможна використовувати вирази, що повертають числові значення. Якщо ви заглянете в таблицю пріоритетів,ведену в розділі 2, то побачите, що дужки є абсолютно необхідними, проте вони допомагають зробити вирази ясніше.

Перше порівняння є істинним, тому повертаєbool-значення true. Зміннау містить дуже мале негативне число, -0,000000000025, тому воно більше, ніж -1.

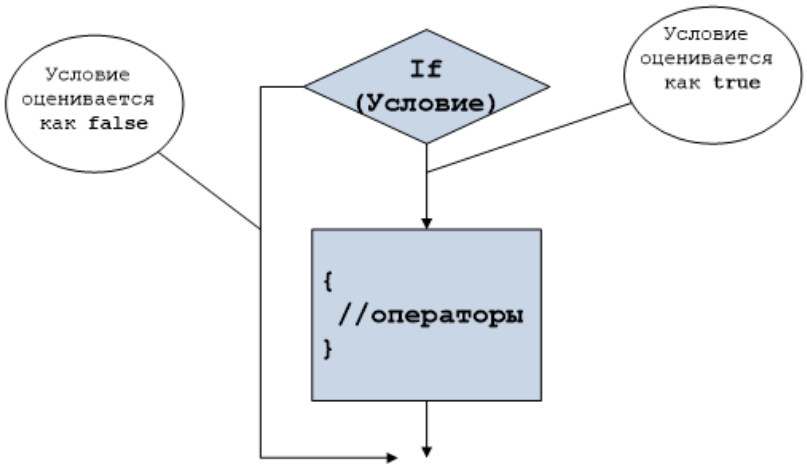
Друге порівняння повертає false. Вираз 10 - і дорівнює 20, тобто тому ж, що j.

Третє вираз повертає true, тому що 3 + у трохи менше, ніж 3.

Ви можете використовувати операції відносин для порівняння значень будь-якогофундаментального типу, тому все, що вам потрібно, — якийсь практичний спосіб використання результатів порівняння для модифікації поведінки програми.

Оператор if

Базовий оператор if дозволяє програмувати виконання єдиного оператора або блоку операторів, укладених у фігурні дужки, якщо цей умовний вираз оцінено як істинне, або пропустити оператор або блок операторів, якщо умова оцінено як помилкове. Це показано малюнку.



Приклади

```
if (letter == 'A')
        cout << “first letter”;
```

Перевірена умова поміщається в дужки, що йдуть за ключовим словом if, після чого слідує оператор, який повинен бути виконаний, якщо умова повертає true. Крапка з комою йде після наступного за if зі дужками оператора. Оператор зрушений, щоб відзначити, що він повинен бути виконаний тільки в тому випадку, коли умова поверне true.

Можна розширити приклад.

```
if (letter == 'A')
{       cout << “first letter”;
        letter = 'a';
}
```

Оператори блоку виконуються при умові true. У блоці може бути необхідна кількість операторів. Без дужок тільки перший вираз був би суб'єктом if.

Вкладені оператори if

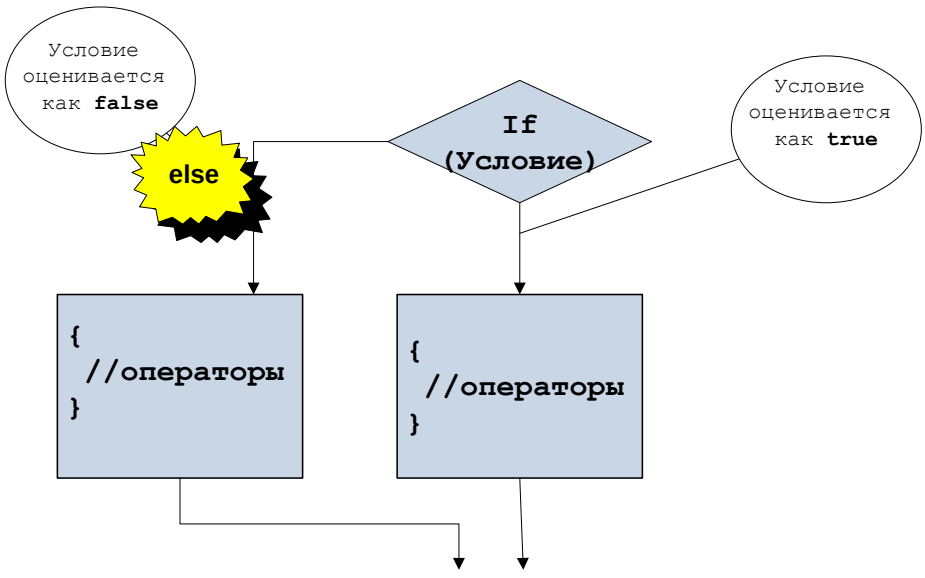
Оператор, який має бути виконаний у разі істинності умови if, може бути ще одним оператором if. Така організація називається вкладеною if. Вкладений if, у свою чергу, може містити ще один вкладений if.

```
if (a > 3)
        if (a <10)
                cout <<” a belongs to the interval 3 -10”;
cout <<” a does not belongs to the interval”;
```

Розширений оператор if

Ця версія if дозволяє виконати один оператор, коли if повертає true, та інший, коли воно повертає false. Після цього виконання програми продовжується з наступного оператора по порядку.

оператор If-else



```
// Приклад: Визначити знак числа
// Використання розширеного оператора if
#include <iostream>
using namespace std;
void main()
{       int a;
        cout << "input number\n";
        cin >> a;
        if (a>=0)
                cout <<" number positive" <<a<<"\n";
        else cout << "namber negative" <<a<<"\n";
}
//Приклад: Визначити парність числа
// Використання розширеного оператора if
#include <math.h>
#include <iostream>
using namespace std;
int main()
{       long number = 0L;
        cout << endl;
                << “ input integer number less than two milliards”<<endl;
        cin >> number;
        if (number % 2L)
                cout << endl<< “number odd “<<endl;
        else
                cout << endl
                        << “number even “
                        <<endl;
        return 0;
}
```

Після читання вхідного значення number, воно перевіряється на парність шляхом взяття залишку при розподілі на 2 та перевірки його за умови оператора if. У разі умова повертає ціле, а чи не булевское значення. Оператор if інтерпретує ненульове значення, повернене умовою як true, а нульове, як false. Еквівалентно такому (number % 2L! = 0).

Умова операторaif може бути виразом, що повертає значення будь-якого з фундаментальних типів даних. Коли умовний вираз повертає числове значення замість bool, компілятор вставляє автоматичне приведення такого результату до типу bool. Приведення до bool ненульового значення дає true, а нульове false.

```
//Приклад: Реалізація примітивного калькулятора
//Вкладений оператор if
#include <math.h>
#include <iostream>
using namespace std;
void main()
{       double x,y;
        char oper;
        cout <<"input number1 operator number2\n";
        cin >>x>>oper>>y;
        if (oper=='+') cout<<"="<<x+y<<"\n";
                else if (oper=='-') cout<<"="<<xy<<"\n";
                                else if (oper=='*') cout<<"="<<x*y<<"\n";
                                                else if (oper=='/') cout<<"="<<x/y<<"\n";
                                                        else cout <<"unknoun operator\n";
}
```

Вкладені оператори if – else

Можна вкладати if - else всередину операторів if, а оператор if - всередину if - else.
//Приклад:

```
if (coffee == 'y')
        if (donuts == 'y')
                cout << “we have coffee and donuts”;
        else    cout << “we have coffee and have not donuts”;
```

При неправильному структуруванні else можна припуститися помилки.
else завжди відноситься до найближчого попереднього if, у якого немає іншого else.
У разі складного коду використовуються фігурні дужки.
//Приклад:

```
if (coffee == 'y')
{       if (donuts == 'y')
                cout << “we have coffee and donuts”;
        else
                cout << “we have coffee and have not donuts”;
}
```

Тепер розглянемо вкладення if всередину if - else.

```
if (coffee == 'y')
{
        if (donuts == 'y')
```

```
                cout << “we have coffee and donuts”;
}
else
{
        if (donuts == 'y')
                cout << “we have not coffee and have donuts”;
}
```

А тепер розглянемо випадок вкладення if-else в інші оператори if-else.

```
//Приклад:
if (coffee == 'y')
{       if (donuts == 'y')
                cout << “we have coffee and donuts”;
        else
                cout << “we have coffee and have not donuts”;
}
else
{       if (tea == 'y')
                cout << “we have not coffee, but have tea and may be donuts…”;
        else
                cout << “we have not coffee, have not tea but may be some donuts…”;
}
```

Потрібності у фігурних дужках немає, але код виглядає ясніше.

Логічні оператори І, АБО, НЕ.

В той час якоператори порівняннявикористовуються для перевірки конкретної умови: хибне воно чи дійсне, вони можуть перевірити лише одну умову за певний проміжок часу. Але бувають ситуації, коли потрібно протестувати одразу кілька умов. Наприклад, щоб дізнатися, чи ми виграли в лотерею, нам потрібно порівняти всі цифри купленого квитка з виграшними. Якщо в лотереї 6 цифр, потрібно виконати 6 порівнянь, всі з яких повинні бути true.

У мові С++ є 3 логічні оператори:

Оператор	Символ	Приклад	Операція
Логічне НЕ	!	!x	true, якщо x - false і false, якщо x - true

Логічне І	&&	x && y	true, якщо x та y — true, інакше — false
Логічне АБО	||	x || y	true, якщо x або y - true, інакше - false

Логічне І

Операція логічного І (&&) застосовується тоді, коли є дві умови та обидва повинні повернути результат true, щоб загальний результат був true.

Таблиця істинності для операції логічного І

	умова 2		
умова 1	&&	false	true
	false	false	false
	true	false	true

//Приклад:

Перевіряємо, чи готовий сніданок

```
if( (coffee == 'y') && (donuts == 'y'))
        cout << “we have coffee and donuts”;
```

Виведення повідомлення відбудеться лише в тому випадку, якщо обидві умови, об'єднані операцією && виявляться дійсними.

Можна перевірити, чи знаходиться number в інтервалі 0< number <1

```
if( (number >0) && (number <1))
        cout << “number is situated in an interval”;
else
        cout << “number is not situed in an interval”;
```

Логічне АБО

Операція логічного АБО(||) застосовується тоді, коли є дві умови і потрібно отримати результат true, якщо одне або обидва повертають true.

Таблиця істинності для операції логічного АБО

	умова 2		
умова 1	||	false	true
	false	false	true
	true	true	true

//Приклад:

Можна перевірити чи знаходиться number поза інтервалом 0-1, тобто. number має бути більше 1 або менше 0.

```
if( (number >1) || (number <0))
        cout << “number is situated outside an interval”;
else
        cout << “number is situated in an interval”;
```

Логічне НЕ

Третя логічна операція –НЕ(!) – приймає лише один операнд bool та інвертує його значення.

Якщо x має значення 10, то вираз! (X> 5) буде false, т.к. x > 5 відповідає true.

І, нарешті, можна застосовувати операцію! до інших базових типів даних. Припустимо, що є змінnarate типу float, що містить значення 3.2.

Через деяку причину може виникнути бажання переконатися. що значення rate відмінно від нуля. Можна використовувати наступний вираз ! (rate).

Значення 3.2 відрізняється від нуля, і тому перетворюється на значення true типу bool, і результат виразу буде false.

Комбінування логічних операцій.

Можна комбінувати умовні висловлювання та логічні операції, як буде зручно. наприклад, можна сконструювати тест , визначальний, чи символ до літер, застосувавши єдиний оператор if.

```
//Приклад:
//Перевірка літери з використанням логічних операцій
#include <iostream>
using namespace std;
int main()
{       char letter = 0;//тут зберегти введення
        cout << endl <<"input symbol:";
        cin >> letter;
        if(((letter >= 'A') && (letter <= 'Z')) ||
                ((letter >= 'a') && (letter <= 'z')))      //Перевірка на
                                                           //Входження в алфавіт
                cout<< endl
                        << " you symbol is letter "
                        << endl;
        else
                cout<< endl
                        << " you symbol is not letter "
                        << endl;
        return 0;
}
```

Перший логічний вираз повертаєtrue, якщо введено букву верхнього регістру, а друге – якщо це буква нижнього регістру.

Якщо виразі є оператори && і ||, то частина операторів у ньому може виконатися. Точніше, їх виконається стільки, скільки потрібно, щоб дізнатися результат всього виразу.

Давайте подивимося на такі логічні вислови:

((i!=0) && (j!=1))

((i! = 0) | | (j! = 1))

У першому вираженні, якщо і дорівнює нулю, тобто перша умова не виконується, то зовсім не важливо, чи j одиниці чи ні. У будь-якому випадку результат всього виразу дорівнюватиме нулю. Так само, якщо у другому виразі і не дорівнює нулю, немає сенсу рахувати далі - в результаті все одно отримаємо одиницю.

Факт, як на мене, дуже важливо знати з двох причин.

По-перше, подивіться на такий вираз:

((i!=0) && (--j!=1))

Залежно від того, чи виконується перша умова, друга умова може бути проігнорована при роботі завдання, а значить, не виконається оператор декременту, що стоїть у ньому.

Оператор множинного вибору switch

в C++ ще є оператор множинного вибору **switch**, який ми зараз розглянемо детально.

```
// Форма запису оператора множинного вибору switch
switch (/*змінна або вираз*/)
{
case /*константний вираз1*/:
{
/*група операторів*/;
break;
}
case /*константний вираз2*/:
{
/*група операторів*/;
break;
}
//. . .
default:
{
/*група операторів*/;
}
}
```

На початковому етапі аналізується вираз чи змінна. Після чого здійснюється перехід до тієї гілки програми, для якої значення змінної або виразу збігається із зазначеним константним виразом. Далі виконується оператор або група операторів доки не зустрінеться зарезервоване слово **break** або закриває фігурна дужка. Якщо значення змінної або виразу не співпадає з жодним константним виразом, то передається управління гілки програми, що

містить зарезервоване слово **default.** Після чого виконується оператор або група операторів цієї галузі.

Розглянемо завдання із використанням оператора вибору **switch.**

//Приклад: Написати програму, яка складає, віднімає, множить, ділить два числа, введені з клавіатури. Розробити інтерфейс користувача.

```
#include <iostream>
using namespace std;
int main()
{	int count; // змінна для вибору switch
	double a,b; // Змінні для зберігання операндів
	cout << "Vvedite pervoe chislo: ";
	cin >> a;
	cout << "Vvedite vtoroe chislo: ";
	cin >> b;
	cout << "Vibirite deistvie: 1-clojenie; 2-vichitanie; 3-ymnojenie; 4-delenie: ";
	cin >> count;
	switch (count) // початок оператора switch
{
	case 1: // якщо count = 1
{
cout << a << " + " << b << " = " << a + b << endl; // Виконати додавання
break;
}
	case 2: // якщо count = 2
{
cout << a << " - " << b << " = " << a - b << endl; // виконати віднімання
break;
}
	case 3: // якщо count = 3
{
cout << a << " * " << b << " = " << a * b << endl; // Виконати множення
break;
}
	case 4: // якщо count = 4
{
cout << a << " / " << b << " = " << a / b << endl; // Виконати розподіл
break;
```

```
}
default: // якщо count дорівнює будь-якому іншому значенню
cout << "Nepravilni vvod" << endl;
}
return 0;
}
```

Якщо ж значення змінної **count** не збігається з жодним константним виразом, то передається управління гілки програми, що містить зарезервоване слово **default.** Тобто буде виконано наступний рядок

```
1        cout << "Nepravilni vvod" << endl;
```

Оператор **switch** може містити, а може й не містити зарезервоване слово **default**. Якщо значення змінної не співпаде з жодним константним виразом і не буде **default**, то програмне управління в цьому випадку просто перейшло б до першого оператора після **switch**.

8. ПОВТОРЕННЯ БЛОКУ ОПЕРАТОРІВ. ЦИКЛ.

Можливість повторно виконувати групу операторів є фундаментальною для більшості програм. Доводиться виконувати одні й самі дії багаторазово. Задля реалізації цієї потреби у мовах програмування передбачені відповідні оператори – вони називаються операторами циклу.

Цикл виконує послідовність операторів до того часу, поки істинно чи хибно певне умова.

Використання циклу for

```
// Приклад:
//Сумування цілих чисел циклом for
#include <iostream>
using namespace std;
intmain()
{       int i = 0, sum = 0;
        const int max = 10;
        for (i = 1; i <= max; i++)
                sum += i;
        cout << endl << "sum=" << sum << endl << "i=" << i << endl;
        return 0;
}
```

Результат виконання програми:

sum = 55

i = 11

Умови, що визначають операцію циклу, з'являються у дужках після ключового слова for. У дужках містяться три вирази, розділені крапкою з комою.

Перший вираз **i = 1** виконується один раз, на початку. та встановлює початкову умову циклу. У разі змінної і присвоюється значення 1.

Другий вираз **i <= max** - логічне - визначає, доки повинен виконуватись оператор циклу (або блок операторів). Якщо друге вираз істинно, цикл продовжує виконуватись; коли воно помилкове, він завершується і виконання програми продовжується з оператора, розташованого за тілом циклу.

В даному випадку оператор циклу в рядку, наступному за for, виконується до тих пір, поки значення і менше або дорівнює max.

Третій вираз **i++** виконується після оператора циклу (або блоку операторів), і в даному випадку воно збільшує і на 1 на кожній ітерації.

Узагальнена форма циклу for виглядає так:

for (вираз ініціалізації; вираз перевірки; вираз інкременту)
оператор усередині циклу;

оператор усередині циклу – може бути окремим оператором чи блоком операторів у фігурних дужках.

Логіку циклу for можна розглянути на наведеній нижче схемі.

Прямокутники позначають події, а ромби – умова.

Варіації циклу for

Вираз ініціалізації може містити оголошення змінної циклу

```
for (int i = 1; i<= max; i++)
        sum+= i;
```

Але в цьому випадку змінна циклу зникає після завершення циклу.

Можна взагалі виключити ініціалізуючий вираз із циклу.

```
int i = 1;
for (; i<= max; i++)
        sum+= i;
```

Але, як і раніше, важлива точка з комою, що відокремлює вираз ініціалізації від перевірочної умови циклу. Фактично обидві точки з комою повинні бути присутніми, незалежно від того, чи пропущено якесь одне з керуючих виразів. інакше компілятор не може зрозуміти, який саме з трьох операторів, що управляють, відсутній.

Цикл for може бути порожнім. Можна помістити оператор циклу у вираз інкремента.

```
for (i = 1; i<= max; sum += i++);
```

Точка з комою потрібна, щоб вказати, що оператор циклу порожній.

У кожен із розділів for можна вставити стільки виразів, скільки знадобиться.

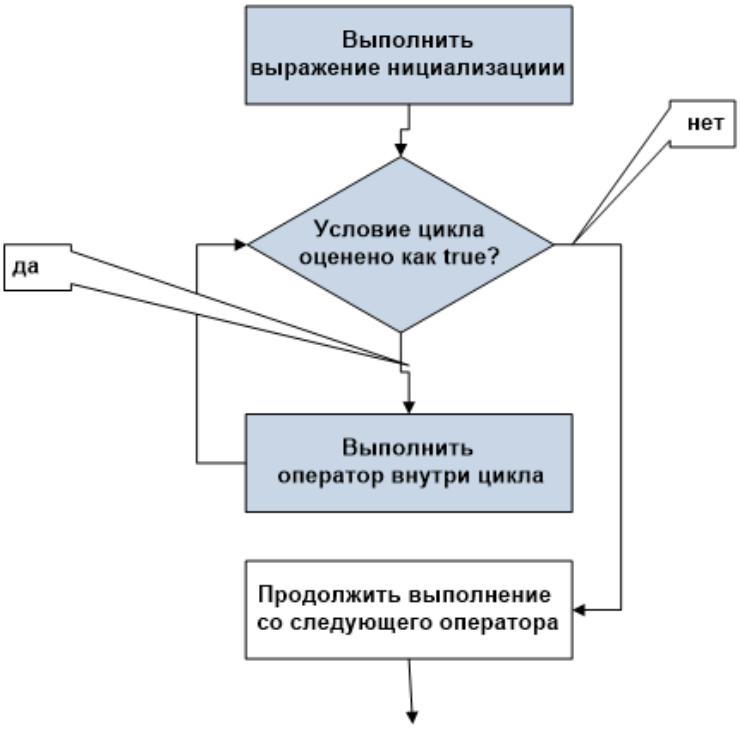
```
for (i = 0,power = 1; i<= max; i++, power += power);
```

Цикл while

Другий тип циклів у C++ цикл while. У той час як цикл for призначений головним чином для повторення оператора або блоку операторів певну кількість разів, цикл while служить для виконання оператора або блоку, поки зазначена умова залишається істинною. Загальна форма циклу while:

while (умова)
оператор усередині циклу;

Тут оператор всередині циклу виконується повторно доти, доки вираз умова має значення true. Після того, як умова стає рівною false, програма виходить з циклу і переходить до оператора, що йде за ним. Єдиний оператор усередині циклу може бути замінений блоком операторів у фігурних дужках.



// Приклад: підсумовування чисел за допомогою циклу while

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

```
int main()
{       inti = 1, sum = 0;
        const int max = 10;
        while(i <= max)
        {       sum += i;
                ++i;
        }
                cout << endl
                        << "sum=" << sum
                        << endl
                        << "i=" << i
                        << endl;
                return 0;
}
```

//Використання циклу while для обчислення середнього арифметичного

```
#include <iostream>
using namespace std;
intmain()
{       double value = 0.0;                     //Введене значення
        double sum = 0.0;                       //сума значень
        int i = 0;                              //лічильник значень
        char indicator = 'y';                   // Значення індикатора продовження
        while (indicator == 'y')
        {       cout  <<  endl <<  "input number:";
                cin >> value;                   //читати значення
                ++i;                            //збільшити лічильник
                sum = sum + value;              //додати поточне значення до суми
                cout << endl
                        << "input number (for finish input n)";
                cin >> indicator;               //читати індикатор
        }
        cout << endl
                << " AV from " << i
                << endl
                << sum/i
                << endl;
```

```
        return 0;
}
//Результат виконання програми
input number:5
input number (for finish input n)y
input number:6
input number (for finish input n)y
input number:7
input number (for finish input n)y
input number:8
input number (for finish input n)n
 AV from 4
6.5
```

Цикл do-while

Циклdo-while подібний до циклу while в тому, що він виконується доти, поки зазначена умова залишається істинною. Головна відмінність полягає в тому, що тут умова перевіряється наприкінці циклу – що відрізняє його від циклів while та for, де умова перевіряється на початку.

Загальна форма циклу do-while:

```
do
{
        оператор усередині циклу;
}while (умова);
```

логіка цієї форми циклу представлена малюнку

//Використання циклу do-while для обчислення середнього арифметичного

```
#include <iostream>
```

```
using namespace std;
intmain()
{       double value = 0.0;                     //Введене значення
        double sum = 0.0;                       //сума значень
        int i = 0;                              //лічильник значень
        char indicator = 'y';           // Значення індикатора продовження
        do
        {       cout  <<  endl << "input number:";
                cin >> value;                           //читати значення
                ++i;                                    //збільшити лічильник
                sum = sum + value;              //додати поточне значення до суми
                cout << endl   << "input number (for finish input n)";
                cin >> indicator;                       //читати індикатор
        } while (indicator == 'y');
        cout << endl
                << " AV from " << i
                << endl
                << sum/i
                << endl;
        return 0;
}
//  Результат виконання програми
input number:5
input number (for finish input n)y
input number:6
input number (for finish input n)y
input number:7
input number (for finish input n)y
input number:8
input number (for finish input n)y
input number:9
input number (for finish input n)n
AV from 5
7
```

Між двома останніми версіями циклу немає особливої різниці, крім того, що правильна робота цієї версії не залежить від початкового значення змінної індикатора.

Вкладені цикли

Цикли можна вкладати один одного. Така техніка зазвичай застосовується для повторення дій різних рівнях класифікації. Про це трохи згодом.

```
//Приклад: демонстрація вкладених циклів для обчислення факторіалу
#include <iostream>
using namespace std;
intmain()
{       intvalue = 0,
        factorial = 0;
        char indicator = 'n';
        do
        {       cout << endl   << "input number:";
                cin >> value;
                factorial = 1;
                for(int i = 2; i <= value; i++)
                        factorial *= i;
                cout << endl
                        << "factorial" <<value<< "=" <<factorial;
                cout << endl
                        << "do you want input number (y or n)?";
                cin >> indicator;
        } while (indicator == 'y');
        return 0;
}
//Результат виконання програми:
input number:3

factorial 3 = 6
do you want input number (y or n)?y

input number:5

factorial 5 = 120
do you want input number (y or n)?y
input number:7
```

factorial 7 = 5040

do you want input number (y or n)?n

Зовнішній із двох циклів , do-while , управляє завершенням програми, до того часу, поки вводиться у у відповідь запит про продовженні, програма продовжує обчислювати значення факторіалу. Чинник цілих чисел обчислюється у вкладеному циклі for. Він виконується, раз множивши змінну factorial (чиє початкове значення дорівнює 1), на послідовні цілі числа – від 2 до value.

Безкінечний цикл for. Оператор break.

Якщо пропустити другий керуючий вираз, який визначає перевірну умову циклуfor, то передбачається, що умова завжди буде true, тому цикл буде продовжуватися нескінченно, якщо не передбачити будь-який інший спосіб виходу з нього. Насправді можна пропустити всі вирази в дужках після for.

//Приклад: використання нескінченного циклу для обчислення середнього

```
#include <iostream>
using namespace std;
intmain()
{       double value = 0.0;                     //Введене значення
        double sum = 0.0;                       //Сума значень
        int i = 0;                              //Лічильник значень
        char indicator = 'n';                   //Індикатор продовження
        for(;;)
        {       cout << endl   << "input number:";
                cin >> value;                   //Читати значення
                ++i;                            //Збільшити лічильник
                sum += value;                   //Додати поточне до суми
                cout << endl
                        << "do you want input number (для finish press n)?:";
                cin >> indicator;               //Читати індикатор
                if((indicator =='n') || (indicator =='N'))
                        break;                  //Вихід із циклу
                        else continue;
        }
                cout << endl
                        << "AV from " << i
                        << "entered values equal " << sum/i
                        << endl;
                return 0;
        }
```

//Результат виконання програми:

input number:10

do you want input number (для finish press n)?:y

input number:20

do you want input number (для finish press n)?:y

input number:40

do you want input number (для finish press n)?:y

input number:50

do you want input number (для finish press n)?:n

AV from 4entered values equal 30

Ця програма обчислює середню величину довільного числа значень. Після введення кожного числа користувач повинен вказати, чи бажає ввести ще одне значення, ввівши символ y або n.

Після оголошення та ініціалізації змінних, які будуть використані, запускається циклfor без керуючих виразів, а отже без гарантій завершення. Безпосередньо наступний блок – суб'єкт циклу, який має повторюватися.

Блок циклу виконує такі дії:

– Читає значення.
– Додає прочитане з cin значення sum.
– Перевіряє, чи хочете продовжити введення значень.

Перша дія всередині блоку запрошує ввести число та читає введене значення у змінnyvalue. Ведене значення додається до sum, після чого задається питання про продовження. Пропонується ввести символ n, якщо продовження є небажаним. Введений символ міститься в змінну indicator для подальшої перевірки на рівність n або N. Якщо рівності немає – цикл триває, інакше виконується break.

Він призводить до негайного виходу з циклу з передачею управління оператору, що йде за фігурною дужкою блоку циклу, що закривається.

Далі виконується виведення кількості введених значень та їхньої середньої величини.

Використання оператора continue

Є ще один оператор крім break, що служить керувати циклом – continue. Записується він дуже просто:

continue;

Виконання continue всередині циклу негайно починає наступну ітерацію циклу, пропускаючи оператори тіла циклу, які залишалося виконати у поточній операції. Можна продемонструвати, як це працює, на наступному прикладі:

//Приклад: використання оператора continue

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

```
        int i = 0; value = 0; product = 1;
        for (i = 1; i <= 10; i++)
        {
                cout << "input integer number: ";
                cin >> value;
                if (value == 0)
                        continue;
                product *= value;
        }
        cout << "result: " << product << endl;
        return 0;
}
```

//Результат виконання програми:

```
input integer number: 4
input integer number: 5
input integer number: 6
input integer number: 0
input integer number: 7
input integer number: 8
input integer number: 9
input integer number: 10
input integer number: 11
input integer number: 12
result: 79833600
```

Цикл читає 10 чисел із наміром отримати добуток всіх введених значень. Операторif перевіряє, чи не було введено 0, і, якщо це так, то виконується continue, щоб відразу перейти до наступної ітерації. Це робиться для того, щоб добуток введених чисел не звернувся в нуль при першому введенні нульового значення. Зрозуміло, якщо нуль буде введено на останній ітерації, цикл просто завершиться. Безумовно, існують інші способи досягти того ж результату, але continue представляє дуже зручний засіб, зокрема, для складних циклів, де вам може знадобитися пропускати залишок поточної ітерації, починаючи з різних точок його тіла.

9. МАСИВИ.

Масив– це сукупність даних, яка має такі властивості:

- всі елементи масиву мають той самий тип;
- масив має одне ім'я всім елементів;

– доступ до конкретного елементу масиву здійснюється за індексом (індексом).

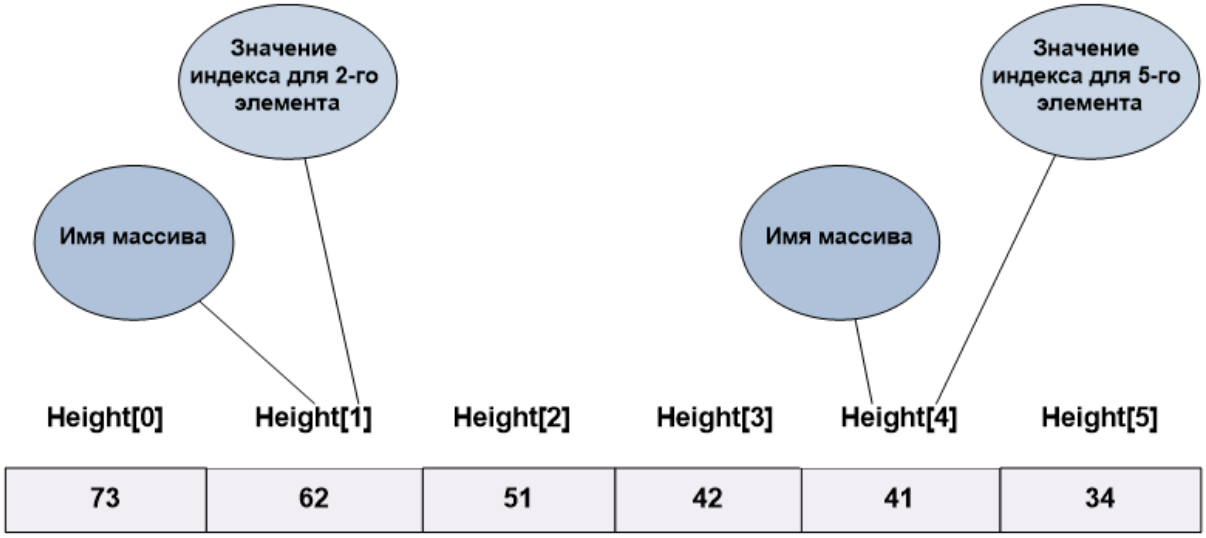
Оголошення масиву має наступний синтаксис:

<специфікація типу> <ім'я> [<константний вираз>];

<специфікація типу> <ім'я> [ ];

Оголошення масиву може мати одну із двох синтаксичних форм, зазначених вище. Квадратні дужки, що йдуть за ім'ям, – ознака того, що змінна є масивом. Константний вираз, укладений у квадратні дужки, визначає число елементів у масиві. Індексація елементів масиву мовою C++ починається з нуля. Таким чином, останній елемент масиву має індекс на одиницю менше, ніж кількість елементів масиву.

Базова структура масиву:



На малюнку показано масив. Ім'я height міститьшість елементів, кожен містить окреме значення. Це може бути значення зростання членів сім'ї, виміряні в дюймах. Оскільки є шість елементів значення індексу знаходиться в межах від 0 до 5.

Щоб послатись на певний елемент, ви пишете ім'я масиву, за яким йде значення індексу певного елемента, укладене у квадратні дужки. Так, наприклад, щоб послатися третій елемент, потрібно написати height[2]. Якщо ви уявите індекс у вигляді зміщення від першого елемента, то легко побачити, що значення індексу четвертого елемента буде рівним 3.

Загальнийобсяг пам'яті, необхідний зберігання кожного елемента, визначається його типом, і всі елементи масиву зберігаються у одному безперервному блоці пам'яті.

Оголошення масивів

По суті, ви оголошуєтемасив точно так, як і досі оголошували змінні. Єдиною відмінністю є те, що поряд з ім'ям масиву слід зазначити кількість його елементів. Наприклад, ви можете оголосити масив цілих чисел height, показаний на малюнку, використовуючи наступний оператор оголошення:

long height[6]

Оскільки кожнезначення типу long займає 4 байти пам'яті, весь масив потребує 24 байт. Масиви можуть мати будь-який розмір - кількість елементів обмежена лише обсягом доступної пам'яті комп'ютера, де виконується ваша програма.

Ви можете оголосити масив будь-якого типу. Наприклад, щоб оголосити масив, призначений для зберігання робочого об'єму та вихідної потужності серії двигунів, ви можете написати таке:

```
double cubic_inches[10]; // Обсяги двигунів у кубічних дюймах
double horsepower[10]; // Вихідні потужності
```

Таким чином можна зберігати робочі об'єми і потужності до 10 двигунів, посилаючись на них за індексами від 0 до 9. Як ви вже бачили це на прикладі інших змінних, можна оголошувати кілька масивів одного і того ж типу в одному операторі, але на практиці майже завжди краще оголошувати ці змінні окремих операторах.

```
double power[10];
int mas[10];
```

Ініціалізація масивів

Щоб ініціалізувати масив при його оголошенні, потрібно помістити початкові значення елементів в обмежений фігурними дужками список, розділений комами, який знаходиться праворуч від знаку рівності, що слідує за ім'ям масиву.

Приклад:

```
int array[5] = {200,250,300,350,400};
```

Масив називаєтьсяarray містить 5 елементів, кожен з яких зберігає значення типу int. Елементи масиву ініціалізовані списком значень у фігурних дужках, що відповідає послідовності значень індексу масиву, тому в даному випадку array[0] набуде значення 200, array[1] набуде значення 250, array[2] набуде значення 300 і т.д.

Не можна помістити у дужках більше значень, ніж оголошено елементів у масиві, але вказати менше можна. Якщо встановлено менше значень, всі вони присвоюються послідовно елементам масиву, починаючи з першого – тобто. елемента з індексом 0. Елементи масиву, для яких не вказано початкові значення, ініціалізуються нулями. Це не те саме, що відбувається, якщо взагалі не вказати списку ініціалізації. Без списку ініціалізації елементи масиву набувають довільних випадкових значень. Крім того, якщо ви включаєте список ініціалізації, у ньому повинен бути присутнім хоча б один елемент, інакше компілятор генерує повідомлення про помилку.

```
//Приклад: демонстрація ініціалізації масиву
#include <iostream>
#include <iomanip>
using namespace std;
intmain()
{       int value[5] = {1,2,3};
        int junk [5];
        int i;
        cout << endl;
        for (i = 0; i < 5; i++)
                cout << setw(12) << value[i];
        cout << endl;
        for (i = 0; i < 5; i++)
                cout << setw(12) << junk [i];
```

```
        cout << endl;
        return 0;
}
```

Результат виконання програми:

```
 1 2 3 0 0
 -858993460 -858993460 -858993460 -858993460 -858993460
```

У цьому прикладі оголошуються два масиви. Масивvalue ініціалізований частково, а другий- junk, не ініціалізується взагалі.

Перші три елементи масиву value ініціалізуються зазначеними значеннями, а решта – за замовчуванням нулями. У випадку junk всі значення є випадковими, оскільки ніяких початкових значень не вказано взагалі. Елементи цього масиву містять сміття.

Зручний спосіб ініціалізації всього масиву нульовими значеннями – це просто вказати єдине початкове значення 0.

```
        int data[100] = {0};
```

Цей оператор повідомляє масив data розміром 100 елементів, ініціалізованих нулями. Перший елемент ініціалізується значенням, вказаним у фігурних дужках , а решта ініціалізуються нулями за умовчанням, оскільки значення для них не вказано явно.

Можна також опустити розмір масиву числового типу, якщо вказати список значень ініціалізації. Кількість елементів у масиві визначається кількістю зазначених значень, що ініціалізують.

Оголошення масиву

```
        int value [] = {2,3,4};
```

визначає масив із трьох елементів з початковими значеннями 2,3 та 4.

Звичайно, якщо розмір масиву великий, то у такий спосіб виконувати ініціалізацію незручно. У разі для введення початкових значень елементів масиву можна скористатися циклом.

//Приклад: демонстрація ініціалізації масиву з використанням циклу

```
#include <iostream>
#include <iomanip>
using namespace std;
intmain()
{       int value[10];
        int i;
        for (i = 0; i < 10; i++)
        {       cout << "input element" << endl;
                cin >> value[i];
        }
        for (i = 0; i < 10; i++)
                cout << setw(4) << value[i];
        cout << endl;
```

```
        return 0;
}
```

Результат виконання програми:

```
input element
1
input element
2
input element
3
input element
4
input element
5
input element
6
input element
7
input element
8
input element
9
input element
10
 1 2 3 4 5 6 7 8 9 10
```

У цьому прикладі оголошено масив value, ініціалізація якого відбувається в циклі з використанням оператора cin. Значення елементів ми ставимо самі, різні при кожному запуску програми.

Так ініціалізувати масив за його значних розмірів також дуже зручно. Часто виникає потреба ініціалізації масиву випадковими числами.

```
//Приклад: демонстрація ініціалізації масиву  випадковими числами
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <time.h>
using namespace std;
intmain()
{       // Ініціалізація генератора випадкових чисел
        srand((unsigned)time(NULL));

        intvalue[25];
        int i;
        for(i = 0; i < 25; i++)
        {
```

```
            value[i] = rand() % 50;
            cout << setw(4) <<value[i]<< endl;
        }
        return 0;
}
```

Результат виконання програми:
17 37 46 8 33 15 20 18 9 15 31 24 41 19 1 22
17 8 44 8 5 30 22 17 39

У цьому прикладі масив ініціалізується випадковими числами за допомогою генератора випадкових чисел, функції **rand()** .Перед використанням, його необхідно проініціалізувати. Для цього використовується функція **srand((unsigned)time(NULL))**;

Як видно, ініціалізація виконана таймером, відповідна функція є аргументом функції srand() .

Тепер кожному елементу масиву привласнюється значення, що дорівнює залишку від поділу випадкового числа, повернутий генератором випадкових чисел на 50.

Зрозуміло, що елементи масиву в такому разі отримають значення від 0 до 49 включно. Для використання таймера та генератора випадкових чисел необхідно приєднати заголовні файли time.h та stdlib.h.

Багатовимірні масиви

Масиви, які ми визначали досі, мають один індекс та називаються одномірними масивами. Але масив може мати і більше одного індексного Значення-в цьому випадку він називається багатовимірним масивом.

Приклад:

Припустимо, що у вас є поле, на якому розташована плантація бобів, по 10 рослин на грядці, і це поле містить 12 таких грядок (тобто є 120 одиниць рослин). Ви можете оголосити масив для запису ваги бобів, зібраних від кожної рослини, використовуючи наступний оператор:

```
double beans[12] [10];
```

В даному випадкуоголошуєтьсядвомірний масив beans, де перший індекс – номер грядки, а другий – номер рослини на грядці. Щоб звернутися до будь-кого конкретному елементу масиву необхідно вказати два індекси. Наприклад, ви можете встановити значення елемента, що відповідає п'ятій рослині в третій грядці, наступним оператором:

```
beans[2][4] = 10.7;
```

Нагадаємо, щозначення індексів починаються з нуля, тому значенняіндексу третьої грядки-2, а індекс п'ятої рослини - 4.

Будучи успішним бобовим фермером, ви можете мати кілька полів, засіянихбобами за тим самим шаблоном. Припускаючи, що маєте 8 полів, можна визначити тривимірний масив для запису даних про врожай наступним чином:

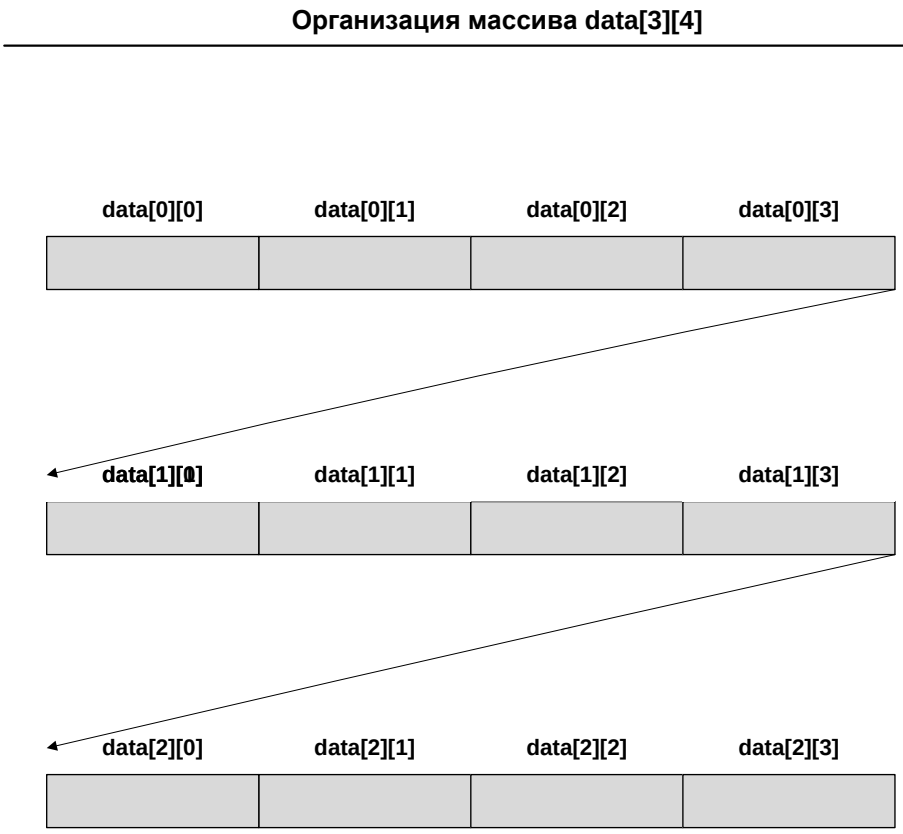
```
double beans[8][12] [10];
```

Така запис дозволяє організуватиоблік кожної рослини, що росте на всіх цих полях, причому найлівіший індекс посилається на певне поле. Якщо ви коли-небудь займетесь вирощуванням бобів у міжнародному масштабі, то зможете використовувати чотиривимірний масив, де додатковий вимір позначатиме країну.

Масиви зберігаються в пам'яті так, що самийправий індекс зростає найшвидше. Тобто масив data [3] [4] це двовимірний масив, що складається з масивів по чотири елементи в кожному. Організація такого масиву показано малюнку.

Елементи масиву розташовуються у безперервному блоці пам'яті, як показано стрілочками малюнку. Перший індекс вибирає певний рядок усередині масиву, а другий індекс вибирає елемент усередині рядка.

Зверніть увагу, що двомірний масив у рідному C++ насправді одновимірний масив, що складається з одновимірних масивів. Масив із трьома вимірами в C++ це одномірний масив елементів, у якому кожен елемент є одномірним масивом одномірних масивів.



Ініціалізація багатовимірних масивів

Для щоб ініціалізувати багатовимірний масив, використовується розширений метод ініціалізації одновимірних масивів.

Наприклад, можна ініціалізувати двовимірний масив data за допомогою наступного оголошення:

```
long data[2][4] {
                    {1, 2, 3, 5},
                    { 7, 11, 13, 17}
                };
```

Тобто, ініціалізаціязначень кожного рядка масиву міститься всередині своєї пари фігурних дужок. Оскільки в кожному рядку чотири елементи, в кожній групі присутні по чотири значення ініціалізації, і оскільки рядків всього два, всередині дужок знаходиться дві групи, що ініціалізують значення. Ви можете пропустити ініціалізацію значень у будь-якому рядку, у цьому випадку решта елементів масиву ініціалізується нульовими значеннями, наприклад:

```
long data[2] [4] = {
                    { 1, 2, 3       },
                    { 7, 11 }
                    };
```

Списки ініціалізації доповнені пробілами, щоб показати де пропущені значення. Елементидata [0][3], data [1][2] і data [1][3] не набувають ініціалізуючих значень і тому дорівнюють нулю.

Якщо ви хочете ініціалізувати весь масив нульовими значеннями, можна просто написати:

```
long data[2] [4] = {0};
```

//Приклад: демонстрація ініціалізації двовимірного масиву випадковими числами

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <time.h>
using namespace std;
intmain()
{       srand((unsigned)time(NULL)); // Ініціалізація генератора випадкових чисел
        intvalue[10][10];
        inti, j;
        for (i = 0; i < 10; i++)
        {       for(j = 0; j < 10; j++)
                {       value[i][j] = rand() % 50;
                        cout <<setw(5)<<value[i][j];
                }
                cout << endl;
        }
                cout << endl;
        return 0;
}
```

Результат виконання програми:

```
31 2 43 7 28 44 12 47 21 47
31 47 27 25 8 8 31 22 21 44
25 37 33 34 32 8 40 33 9 3
23 21 37 3 26 45 14 42 34 19
32 30 38 22 18 8 48 1 8 26
8 41 44 25 34 46 4 43 44 1
29 2 17 31 0 49 39 35 1 39
20 48 45 20 20 8 2 13 43 45
44 36 34 7 33 38 43 37 1 5
29 23 47 7 39 43 26 38 23 23
```

У цьому прикладі двовимірний масив ініціалізується випадковими числами, аналогічно до завдання в попередньому прикладі. Тільки двомірного масиву потрібно два циклу, зовнішній цикл- по рядках, внутрішній цикл –по стовпцям.

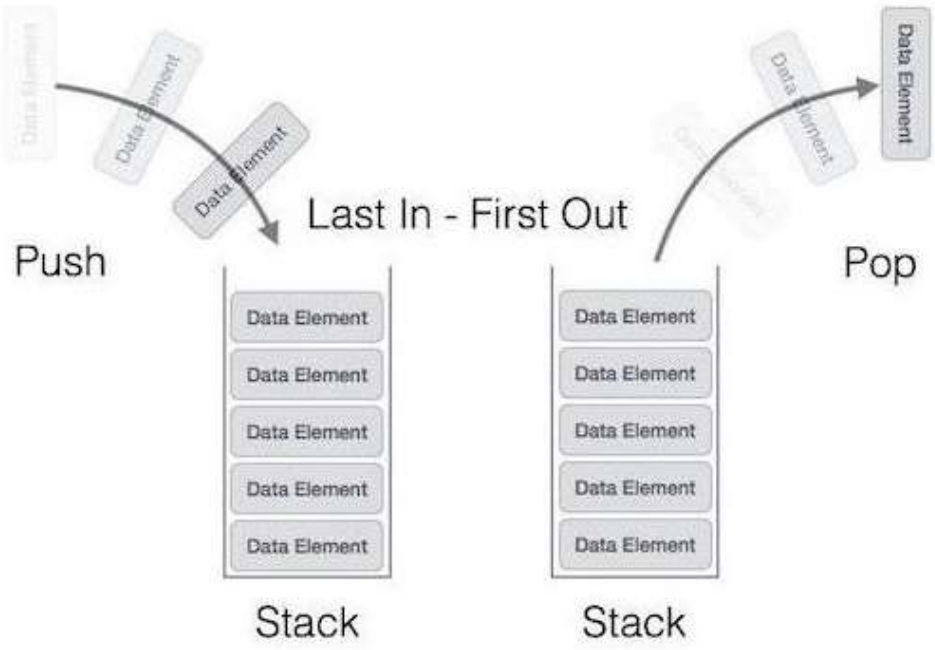
При одному значенні змінної зовнішнього циклу відбувається перебір всіх значень змінної внутрішнього. У даному випадку ми перебираємо всі значення елементів

двовимірного масиву value [10] [10j] по рядках- спочатку всі елементи першого рядка, потім другий і т.д. Після заповнення елемента масиву випадковим числом ми відразу виводимо це значення на екран.

10. ДИНАМІЧНЕ ВИДІЛЕННЯ ПАМ'ЯТІ.

Стек та особливості його роботи.

Стек - це метод представлення однотипних даних у порядку LIFO (Last In - First Out, "перший увійшов - останній вийшов").



Стек - це область оперативної пам'яті, що створюється для кожного потоку. І останній доданий у стек шматочок пам'яті і буде першим у черзі, тобто першим на виведення зі стеку. І щоразу, коли функцією оголошується змінна, вона передусім додається в стек. А коли ця змінна зникає з нашої області видимості (наприклад, функція закінчується), ця змінна автоматично видаляється з стека. У цьому якщо стекова змінна

звільняється, те й область пам'яті, своєю чергою, стає доступною та вільної інших стічних змінних.

Завдяки природі, яку маєстек, управління пам'яттю стає дуже простим і логічним виконання на центральному процесорі. Це підвищує швидкість і швидкодія ЦП, і так відбувається тому, що час циклу оновлення байта дуже незначно (цей байт, швидше за все, прив'язаний до кешу центрального процесора).

Проте у цієї досить строгої форми управління є свої недоліки. Наприклад, розмір стеку - цевеличина фіксована, у результаті при перевищенні ліміту пам'яті, виділеної на стеку, відбудеться переповнення стека.

Об'єм пам'яті стека в програмі, як правило, невеликий: у Visual Studio він дорівнює 1МБ. Якщо ви перевищите це значення, то відбудеться переповнення стека і операційна система автоматично завершить виконання вашої програми.

Як правило, розмір задається під час створення потоку плюс у кожної змінної є максимальний розмір, який залежить від типу даних. Все це дозволяє обмежувати розміри деяких змінних (припустимо, цілісних).

З іншого боку, це змушує оголошувати обсяг складніших типів даних (наприклад, масивів) заздалегідь, оскільки стек не дозволить потім змінити його. До того ж, змінні, які розташовані на стеку, завжди локальні.

Купа.

Купа – сховище пам'яті, розташоване у ОЗУ. Воно допускає динамічне виділення пам'яті та працює не так, як стек. По суті йдеться про простий склад для ваших змінних. Коли ви виділяєте тут ділянку пам'яті для зберігання, до неї можна звертатися як у потоці, так і у всьому додатку загалом (саме так і визначаються змінні глобального типу). Після завершення роботи програми всі виділені ділянки звільняються.

Розмір купи задається під час запуску програми, однак, на відміну від того, як працює стек, у купі розмір обмежений лише фізично, що дозволяє створювати змінні динамічного типу.

Якщо порівнювати, знову ж таки, про те, як працює стек, то купа функціонує повільніше, тому що змінні розкидані по пам'яті, а не знаходяться вгорі стека. Проте цей факт не зменшує важливості купи, і якщо вам треба працювати з глобальними або динамічними змінними, вона більше підходить. Проте керувати пам'яттю тоді має програміст.

Динамічний виділення пам'яті.

Динамічне виділення пам'яті – це спосіб запиту пам'яті з операційної системи запущеними програмами за необхідності. Ця пам'ять не виділяється з обмеженої пам'яті стека програми, а виділяється з більшого сховища, керованого операційною системою - купи. На сучасних комп'ютерах розмір купи може становити гігабайти пам'яті.

Для динамічного виділення пам'яті однієї змінної використовується оператор new (який повертає адресу виділеного простору):

```
// приклад використання операції new
int *ptrvalue = new int;
//де ptrvalue - покажчик на виділену ділянку пам'яті типу int
```

//new – операція виділення вільної пам'яті під створюваний об'єкт.

Операція new створює об'єкт заданого типу, виділяє йому пам'ять і повертає покажчик правильного типу дану ділянку пам'яті.

Покажчик- Це змінна, значенням якої є адреса осередку пам'яті. Покажчики оголошуються так само, як і звичайні змінні, тільки зі зірочкою між типом даних та ідентифікатором:

int *ptrvalue;

double* pa;

Якщо пам'ять неможливо виділити, наприклад, у разі відсутності вільних ділянок, то повертається нульовий покажчик, тобто покажчик поверне значення 0. Виділення пам'яті можливо під будь-який тип даних: int, float, double, char тощо.

Динамічно виділена пам'ять немає області видимості, тобто. вона залишається виділеною до того часу, доки явно звільнена чи доки ваша програма завершить своє виконання (і операційна система очистить все буфера пам'яті самостійно). Однак вказівники, що використовуються для збереження динамічно виділених адрес пам'яті, дотримуються правил області видимості звичайних змінних. Ця невідповідність може спричинити цікаву поведінку. Наприклад:

void doSomething()

{

int *ptr=new int;

}

Тут ми динамічно виділяємо цілу змінну, але ніколи не звільняємо пам'ять через **delete**. Оскільки покажчики слідують тим самим правилам, як і звичайні змінні, то, коли функція завершить своє виконання,**ptr** вийде із області видимості. Оскільки **ptr** — це єдина змінна, що зберігає адресу динамічно виділеної цілісної змінної, коли **ptr** знищиться - більше не залишиться вказівників на динамічно виділену пам'ять. Це означає, що програма "втратить" адресу динамічно виділеної пам'яті. І, в результаті, цю динамічно виділену цілу змінну не можна буде видалити.

Це називається **витоком пам'яті (memory leak).** Витік пам'яті відбувається, коли ваша програма втрачає адресу деякої динамічно виділеної частини пам'яті (наприклад, змінної або масиву), перш ніж повернути її назад до операційної системи. Коли це відбувається, програма вже не може видалити цю динамічно виділену пам'ять, оскільки вона більше не знає, де вона знаходиться. Операційна система також не може використовувати цю пам'ять, оскільки вважається, що вона, як і раніше, використовується вашою програмою.

Витоку пам'яті з'їдають вільну пам'ять під час виконання програми, зменшуючи кількість доступної пам'яті не тільки для цієї програми, але й для інших програм також. Програми з серйозними проблемами з витоком пам'яті можуть з'їсти всю доступну пам'ять, внаслідок чого весь ваш комп'ютер повільніше працюватиме або навіть відбудеться збій. Тільки після того, як виконання вашої програми завершиться, операційна система зможе очистити та повернути всю пам'ять, яка «вибігла».

Як динамічне виділення пам'яті працює?

На комп'ютері є пам'ять (можливо, більша її частина), яка доступна для використання програмами. Під час запуску програми, ваша операційна система завантажує цю програму в деяку частину цієї пам'яті. І ця пам'ять, що використовується вашою

програмою, розділена на кілька частин, кожна з яких виконує певне завдання. Одна частина містить ваш код, інша використовується для виконання звичайних операцій (відстеження функцій, створення і знищення глобальних і локальних змінних і т.д.). Тим не менш, більшість доступної пам'яті комп'ютера просто перебуває в очікуванні запитів на виділення від програм.

Коли ви динамічно виділяєте пам'ять, ви просите операційну систему зарезервувати частину цієї пам'яті для використання вашою програмою. Якщо ОС може виконати цей запит, то адреса пам'яті повертається назад у вашу програму. З цього моменту і надалі ваша програма зможе використати цю пам'ять, як тільки забажає. Коли ви вже виконали з цією пам'яттю все, що було необхідно, її потрібно повернути назад в операційну систему, для розподілу між іншими запитами.

На відміну від статичного або автоматичного виділення пам'яті, програма самостійно відповідає за запит та зворотне повернення динамічно виділеної пам'яті**.**

Для змінних це виконується за допомогою оператора **delete.**

// приклад використання операції delete:

delete ptrvalue;

// де ptrvalue - покажчик на виділену ділянку пам'яті типу int

// delete - Операція вивільнення пам'яті

Оператор delete насправді нічого не видаляє. Він просто повертає пам'ять, яку було виділено раніше, назад в операційну систему. Потім операційна система може перепризначити цю пам'ять іншому додатку (або цьому знову).

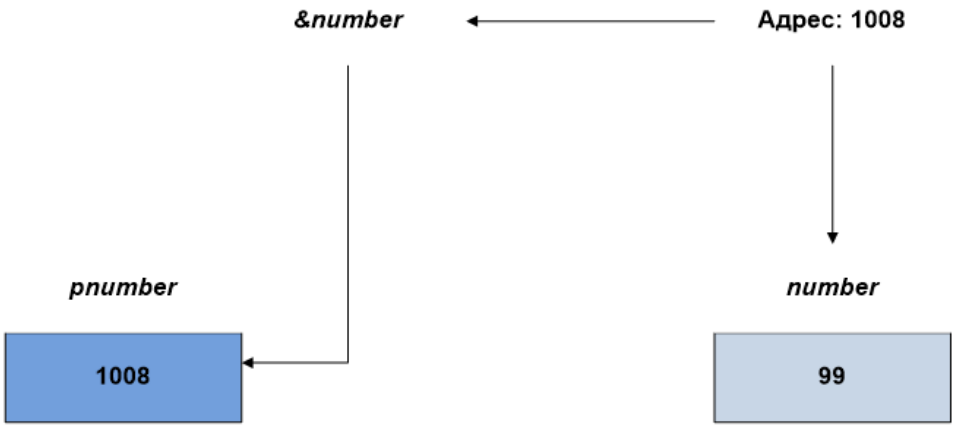
Змінна-покажчик, як і раніше, має ту саму область видимості, що й раніше, і їй можна надати нове значення, як і будь-якій іншій змінній.

Операція отримання адреси

Що вам потрібна для цього – операція отримання адреси, &. Вона представляє зібій унарну операцію, що повертає адресу змінної - свого операнда. Щоб визначити значення адреси змінної вказівнику, ви можете написати наступний оператор присвоєння:

pnumber =&number; // Зберегти адресу number у pnumber

Результат виконання цієї операції можна побачити малюнку.



Ви можете використовувати операцію & дляотримання адреси будь-якої змінної, але для збереження її вам знадобиться змінна-покажчик відповідного типу. Наприклад, якщо

потрібно зберегти адресу змінної double, покажчик повинен бути оголошений з типом double *, тобто "покажчик на double".

Використання вказівників

Отримання адресизмінної та збереження його в змінній-покажчику-це дуже добре, але більш цікаво, як його використовувати. Основний сенс застосування покажчика полягає у можливості доступу до даних, на які він вказує. Це робиться за допомогою операції розіменування (*).

Операція розіменування

Длядоступу до змінної, на яку вказує покажчик, ви застосовуватимете операцію розіменування покажчика (*), підставляючи як операнду ім'я змінної-покажчика. Назва "операція розіменування" або "операція непрямого доступу" говорить про те, що звернення до даних не пряме, і для отримання доступу до даних, на які вказує покажчик, його слід "розіменувати".

Один аспект цієї операції, який може здатися заплутаним, пов'язаний із існуванням різних застосувань одного й того ж символу*. По-перше, це операціяарифметичного множення, по-друге, операція розіменування, також вона використовується при оголошенні змінних-покажчиків. Щоразу, зустрічаючи символ *, компілятор визначає його значення у кожному даному випадку з контексту. Коли ви перемножуєте дві змінні, наприклад, A*B, не існує жодної іншої осмисленої інтерпретації цього виразу, крім операції множення

```
int count1 = 1;
int count2 = 1;
int * pcount;
pcount = &count1;
count2 = * pcount * 10;
```

Створення динамічних масивів

Масиви також можуть бути динамічними. Найчастіше операції new і delete застосовуються створення динамічних масивів, а чи не створення динамічних змінних.

Динамічно виділити пам'ять масиву дуже просто. Якщо необхідно створити масив типу **char**, то припускаючи, що **pString** - покажчик на **char**, можна написати такий вираз:

```
pString = new char [20];      //Виділення пам'яті на 20 символів
```

Ми виділили пам'ять для символьного масиву розміром 20 символів і зберігає вказівник на нього в **pString.**

Щоб звільнити створений таким чином масив та повернути пам'ять у вільне сховище, ви маєте застосувати оператор **delete**. Операція має виглядати так:

```
delete [] pString;       // Видалити масив, який вказує pString
```

Зверніть увагу на квадратні дужки, вони говорять про те, що те, що ви видаляєте – масив. Коли звільняється масив, виділений у вільному сховищі, завжди потрібно вказувати квадратні дужки, інакше результат буде невизначеним. При цьому не потрібно вказувати розмір масиву – до сить просто дужок**[ ]**.

Звичайно, вказівник **pString** після цього буде містити адресу пам'яті, яка може бути вже розподілена для якихось інших цілей, тому, звичайно ж, вона не повинна використовуватися. Коли ви використовуєте оператор **delete** для того, щоб звільнити деяку

пам'ять, яка була виділена раніше, завжди повинні скидати значення вказівника в 0, як показано нижче:

```
pString = 0;   // Встановити покажчик у null
```

Розглянемо, як виділити пам'ять під динамічний динамічний масив.

Нехай розмірність динамічного масиву вводиться із клавіатури. Необхідно виділити пам'ять під цей масив, а потім створений динамічний масив треба видалити. Відповідний фрагмент програми має вигляд:

```
int n;
…
std::cin >> n;                  // Розмірність масиву
int* mas = new int[n];                  // Виділення пам'яті під масив
…
delete[] mas;             //звільнення пам'яті
```

У цьому прикладі mas є вказівником на масив з n елементів.

Вираз

```
int * mas = new int [n];
```

робить дві дії: оголошується змінна типу покажчик, а потім покажчику присвоюється адреса виділеної області пам'яті відповідно до заданого типу об'єкта. Ці дві операції можна поділити.

```
int n;
int* mas;
…
std::cin >> n; // n-розмірність масиву
mas = new int[n]; // Виділення пам'яті під масив
…
delete[] mas;
```

Наприклад розглянемо завдання пошуку суми елементів одновимірного масиву, що складається з n елементів, де розмірність масиву задається у виконання програми. І, відповідно, пам'ять у разі виділяється динамічно.

```
//Приклад: пошук суми елементів динамічного масиву речових чисел
#include <iostream>
void main()
{
        int i, n;               // змінні для індексу та розміру масиву
        float* a;               // покажчика на "float"
        float s;                //змінна для суми
        setlocale(LC_ALL,"Український");
```

```
        std::cout << "Введіть розмір масиву n = ";
        std::cin >> n;              //Введення розміру масиву
        //Виділення пам'яті для динамічного масиву
        a =new float[n];
        std::cout << "Введіть массив A\n";
        for (i = 0; i < n; i++)             //Послідовне введення елементів масиву
                std::cin >>* (a + i);    // можна і так:  cin>>a[i];
        for (s = 0, i = 0; i < n; i++)    //Обчислення суми
                s + = * (a + i);          //s+=a[i];
        std::cout << "S=" <<s<< '\n'; //Виведення результату
        //Звільнення пам'яті
        delete[]a;
}
```

Арифметика вказівників

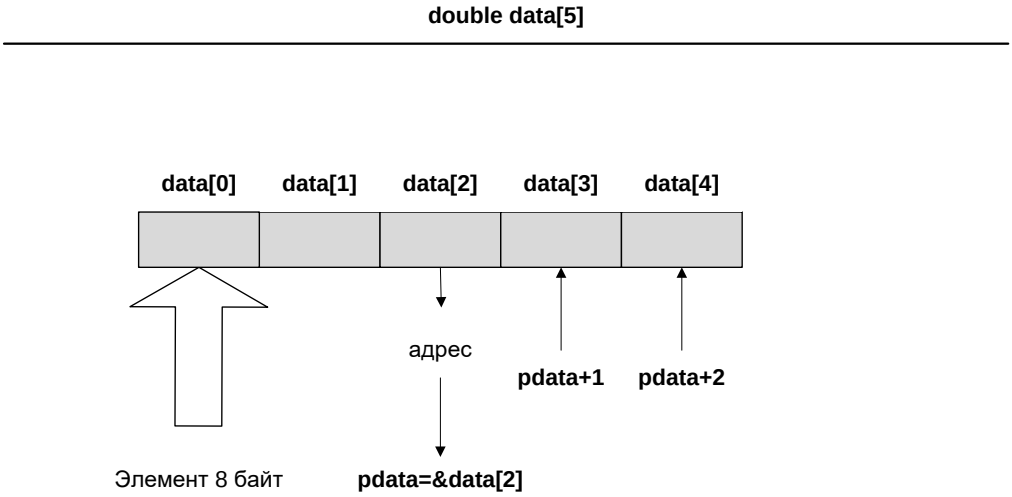
Над вказівниками можна виконувати арифметичні операції. Щоправда, вони обмежені лише додаванням і відніманням, але можна порівнювати значення покажчиків, отримуючи логічний результат. Арифметиканадпокажчиками неявно припускає, що покажчик вказує на масив, і арифметичні операції виконуються над адресою, що міститься в покажчику. Так, наприклад, вказівнику pdata можна присвоїти адресу третього елемента масиву data за допомогою наступного оператора:

pdata = & data[2];

У цьому випадку вираз pdata + 1 посилатиметься на адресу data [3] - четвертого елемента масиву data, тому ви можете переставити вказівник на цей елемент наступним чином:

```
p d a t a   + = 1 ;   / / Інкремент покажчика pdata переносить його на
                        //наступний елемент
```

Цей оператор збільшує адресу, що міститься в pdata, кількість байт, яке займає кожен елемент масиву data.



Іншими словами, інкремент та декремент покажчика працює у термінах типу об'єкта, на який він вказує. Збільшення на одиницю вказівника на long змінює його вміст на адресу

наступного long, тобто збільшує його адресу на вісім. Аналогічно, інкремент покажчика на short на одиницю збільшує значення адреси на дві. Найбільш поширена нотація збільшення покажчика використовує операцію инкремента.

Наприклад:

pdata++; // Збільшитирdata до наступного елемента

Це еквівалентно формі +=, до того ж найчастіше застосовується. Однак, використання форми +=, підкреслює, що хоча зазвичай значення інкременту дорівнює одиниці, ефект від його застосування до покажчика виявляється у збільшенні адреси більше ніж на одиницю, за винятком випадку покажчика на char.

Адреса, отримана в результаті застосування арифметичної операції до покажчика, може змінюватися від адреси першого елемента масиву до адреси, що лежить відразу за останнім елементом. Поза цими межами поведінка покажчика не визначено.

Можна, звісно, розіменувати покажчик, якого застосовано арифметичне дію (інакше у ньому був особливого сенсу).

Наприклад, якщо припустити, що pdata все ще вказує data [2], то оператор:

*** (pdata + 1) = * (pdata + 2);**

еквівалентний наступному:

data[3] = data[4];

Коли ви хочете розіменувати вказівник після збільшення адреси, яку він містить, дужки необхідні, оскільки пріоритет операції розіменування вищий, ніж пріоритет арифметичних операцій + або -. Якщо ви напишете вираз **pdata+1** замість *** (pdata + 1)**, це додасть одиницю до значення, що знаходиться за адресою, що зберігається в pdata, що еквівалентно до виконання **data [2]+ 1.**

Можна використовувати ім'я масиву, як це був покажчик, для звернення до його елементам. Якщо у вас є одномірний масив на зразок того, що раніше, оголошений, як:

long data[5];

то, застосувавши нотацію покажчика, ви можете послатися елемент **data [3]**, наприклад, так: *** (data + 3)**. Цей вид нотації може застосовуватися абсолютно вільно, так що для доступу до елементів **data [0]**, **data [1]**, **data [2]** ви можете писати *** data, * (data +1), * (data +2)** і так далі .

11. ДИНАМІЧНЕ ВИДІЛЕННЯ ПАМ'ЯТІ ДЛЯ БАГАТОВИМІРНИХ МАСИВІВ.

Виділення пам'яті у вільному сховищі для багатовимірного масиву передбачає використання оператора **new** у більш складній формі, ніж для одномірного масиву.

Розглянемо як динамічно виділити пам'ять під двомірний масив.

Нехай потрібно створити двовимірний динамічний масив цілих чисел розмірністю **n** на **k**: ...

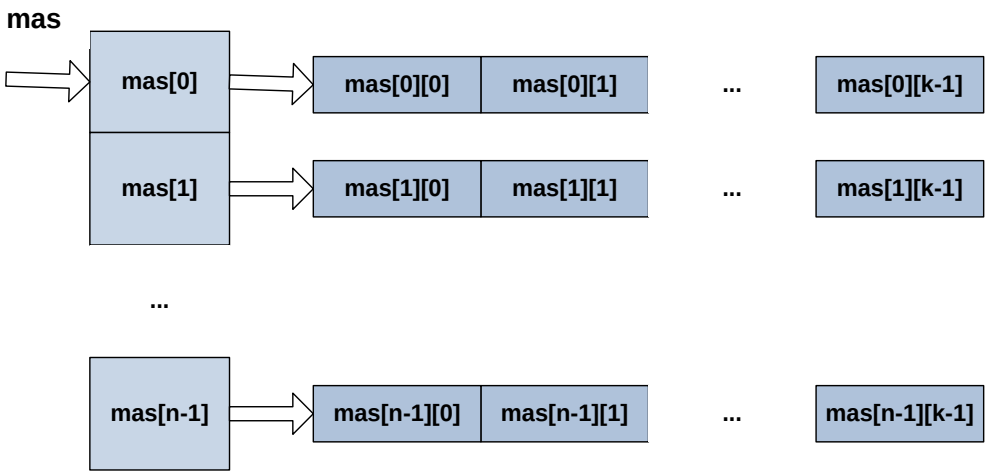
```
int n, k, i;
int** mas;
…
std::cin >> n; //n- число рядків масиву
std::cin >> k;  // k – число стовпців масиву
mas = new int* [n];    //Виділення пам'яті під n покажчиків на рядок
// Виділення пам'яті для кожного рядка за кількістю стовпців k
```

```
for(i = 0; i < n; i++)
        mas[i] = new int[k];
…

for(i = 0; i < n; i++)          // Звільнення пам'яті
        delete mas[i];
delete[] mas;
```

Спочатку необхідно за допомогою оператора **new** виділити пам'ять під **n** вказівників. Виділена пам'ять буде вектором, елементом якого є покажчик. При цьому всі покажчики розміщуються в пам'яті послідовно один за одним. Після цього необхідно в циклі кожному покажчику присвоїти адресу виділеної області розміром, що дорівнює другому кордоні масиву.

Размещение двумерного динамического массива в памяти



А тепер вирішимо задачу матричної алгебри щодо пошуку суми елементів головної діагоналі матриці, із заданими в процесі роботи програми розмірами.

Слід зазначити, що матрицю дуже легко ототожнити з двовимірним масивом. Тому саме завдання матричної алгебри широко використовуються у навчальних цілях для отримання навичок роботи з масивами. Далі в задачах цього типу будемо застосовувати терміни матричної алгебри до двовимірного масиву.

При цьому слід пам'ятати, що нумерація рядків та стовпців масиву починається з 0.

```
//Приклад: визначення суми елементів головної діагоналі динамічного масиву
#include <iostream>
using namespace std;

voidmain()
```

```
{
	int** array;
	int n, m;
	int i, j, sum;
	setlocale(LC_ALL,"Український");
	//Виділення пам'яті для динамічного масиву
	std::cout << "Введіть кількість рядків та стовпців матриці\n";
	std::cin >> n >> m;	//Введення розмірів масиву
	array = new int* [n];	//Виділення пам'яті для масиву покажчиків
	for(i = 0; i < n; i++)	//Виділення пам'яті для рядків масиву
	{	array[i] = new int[m];
		std::cout << "Введіть значення елементів рядка " << i << "\n";
		for (j = 0; j < m; j++) //Введення елементів масиву
			std::cin >> array[i][j];
	}
	//Виведення масиву на екран
	for(i = 0; i < n; i++)
	{
		for (j = 0; j < m; j++)
		{	std::cout << array[i][j] << '\t';	}
			std::cout << endl;
	}
	std::cout << endl;
	// Обчислення суми елементів головної діагоналі
	sum = 0;
	for (i = 0; (i < n) && (i < m); i++)
		sum += array[i][i];

	std::cout << "Сума елементів діагоналі:" << sum << "\n";
	// Звільнення пам'яті
	for(i = 0; i < n; i++)
		delete[] array[i];

	delete[] array;
}
```

У цьому вся прикладі необхідно виділити пам'ять під двомірний масив.

Оголошення **int** array;** вказує на те, що **array** - покажчик на масив покажчиків.

Далі оголошуються змінні **n** і **m** для зберігання розмірів масиву, що вводяться в процесі роботи програми, а також допоміжні індекси та змінна для зберігання суми **i, j, sum**.

У рядках:

```
std::cout << "Введіть кількість рядків та стовпців матриці\n";
std::cin >> n >> m;
```

здійснюється введення розмірів масиву.

Далі виділяється місце у вільному сховищі для динамічного двовимірного масиву. Зверніть увагу, що розмір масиву ми задали під час виконання програми, а чи не так, як задавали її для статичного масиву, за його оголошенні.

Створення двовимірного масиву відбувається у два етапи. Спочатку виділяється пам'ять для масиву покажчиків, за кількістю рядків у масиві, а потім, у циклі, виділяємо пам'ять для кожного рядка масиву. Адреси рядків розміщуються у відповідних елементах масиву покажчиків.

```
array = new int* [n];  //Виділення пам'яті для масиву покажчиків
for(i = 0; i < n; i++)   //Виділення пам'яті для рядків масиву
{
        array[i] = new int[m];
        …
}
```

Після виділення пам'яті для чергового рядка масиву провадиться ініціалізація її елементів шляхом запиту у оператора відповідних значень.

```
std::cout << "Введіть значення елементів рядка " << i << "\n";
for (j = 0; j < m; j++) //Введення елементів масиву
        std::cin >> array[i][j];
```

Для зручності та наочності введені значення виводяться на екран у вигляді відповідної матриці. Це дає можливість оцінити правильність введених даних та отриманого результату.

```
//Виведення масиву на екран
for(i = 0; i < n; i++)
{
        for (j = 0; j < m; j++)
        {
                std::cout << array[i][j] << '\t';
        }
        std::cout <<std::endl;
}
std::cout <<std::endl;
```

Тепер можна підсумувати значення елементів головної діагоналі і вивести результат.

```
        // Обчислення суми елементів головної діагоналі
        sum = 0;
        for (i = 0; (i < n) && (i < m); i++)
                sum += array[i][i];
        std::cout << "Сума елементів діагоналі:" << sum << "\n";
```

І останнім кроком звільняємо пам'ять - спочатку, в циклі, кожного рядка, а вже після цього, масиву покажчиків.

```
    // Звільнення пам'яті
        for(i = 0; i < n; i++)
                delete[] array[i];

        delete[] array;
    }
```

Розглянемо ще одне завдання

```
    // Приклад: дана дійсна матриця m x n. Знайти послідовність,
    //що складається з найбільших значень рядків вихідної матриці

    #include <iostream>
    #include <stdlib.h>
    #include <time.h>
    #include <iomanip>
    using namespace std;
    intmain()
    {
        //Виділення місця для динамічного масиву
        //заповнення масиву
        srand(time(NULL));
        intm, n, i, j;
        int** matr;
        int* b;

        setlocale(LC_ALL,"Український");
        std::cout << "Введіть кількість рядків та стовпців матриці\n";
        std::cin >>m>> n;      //Введення розмірів масиву
```

```
matr =new int* [m];   //Виділення пам'яті під масив покажчиків
std::cout << "m:" <<m<< ", n:" << n << '\n';
std::cout << "Матриця:\n";
for(i = 0; i < m; i++)
{
        matr[i] =new int[n];   //Виділення пам'яті для рядків масиву
//заповнення масиву випадковими числами та виведення масиву на екран
        for(j = 0; j < n; j++)
        {
                matr[i][j] = rand() % 100;
                std::cout <<setw(5)<<matr[i][j];
        }
        std::cout <<std::endl;
}
std::cout <<std::endl;
b =new int[m];                          //масив найбільших значень рядків
for(i = 0; i < m; i++)//Визначення найбільших значень рядків
{
        b[i] = matr[i][0];
        for(j = 1; j < n; j++)
        {
                if(matr[i][j] > b[i])
                {
                        b[i] = matr[i][j];
                }
        }
std::cout <<b[i]<<std::endl;
}
std::cout << "Максимальні значення у рядках:\n";
for(i = 0; i < m; i++) {
        std::cout <<b[i]<<std::endl;
}
// Звільнення пам'яті
for(i = 0; i < m; i++)
        delete[]matr[i];
delete[]matr;
```

```
        delete[]b;
        return 0;
}
```

У цьому прикладі ми знову застосували наші знання програмування на вирішення завдання матричної алгебри.

На відміну від попередніх завдань ми не вводили значення елементів масиву вручну, а скористалися генератором випадкових чисел. Це дуже зручно, особливо якщо припустити, що масив може бути великий, і введення з клавіатури дуже стомлююче.

Щоб працювати з генератором випадкових чисел, потрібно підключити відповідну бібліотеку

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

а в самій програмі скористатися функцією rand() для створення випадкового числа.

Але щоб функція запрацювала, потрібно виконати ініціалізацію генератора випадкових чисел. Ініціалізацію генератора випадкових чисел краще виконати за допомогою системного часу, тоді при кожному новому запуску програми значення в матриці будуть оновлюватися.

Для роботи із системним таймером необхідна бібліотека time.h

```
#include <time.h>
```

Ініціалізація генератора випадкових чисел відбувається у рядку:

```
srand(time(NULL));
```

Повертається генератором число псевдовипадково, і знаходиться у певному, досить великому діапазоні. Для заповнення масиву ми можемо захотіти, і зазвичай хочемо цей діапазон обмежити або підігнати під вирішуване завдання. Так, у разі, береться залишок від цілого розподілу на 100. Тобто. ми отримуємо значення від 0 до 99.

```
matr[i][j] = rand() % 100;
```

Зручності виведення всі значення за допомогою функції setw(5) задаються рівними 5 символів по ширині. Таким чином, незалежно від того виводяться однозначні або двозначні значення, всі вони приймаються рівними 5 символам і результуючий масив виводиться акуратно і без зсувів.

Для зберігання результату оголошується допоміжний масив

```
b = new int [m];
```

Він також створюється динамічно, оскільки його розмір невідомий на початок виконання програми.

Далі визначаються максимальні значення в кожному рядку, і відповідні значення заносяться в масив b.

Перед завершенням програми головне не забути про звільнення усієї виділеної пам'яті.

12. СТРУКТУРНА ОРГАНІЗАЦІЯ ПРОГРАМ. ФУНКЦІЇ.

До цього моменту ви не були готові організовувати код своїх програм у модульному стилі, оскільки могли конструювати програму якєдину функцію – main(); однак ви використовували різноманітні бібліотечні функції, а також функції, що належать об'єктам.

Щоразу, починаючи писати програму на C++, ви повинні продумувати її модульну структуру від початку, і як ви побачите, розуміння того, як повинні бути реалізовані функції, істотно для об'єктно-орієнтованого програмування на C++.

Що таке функції.

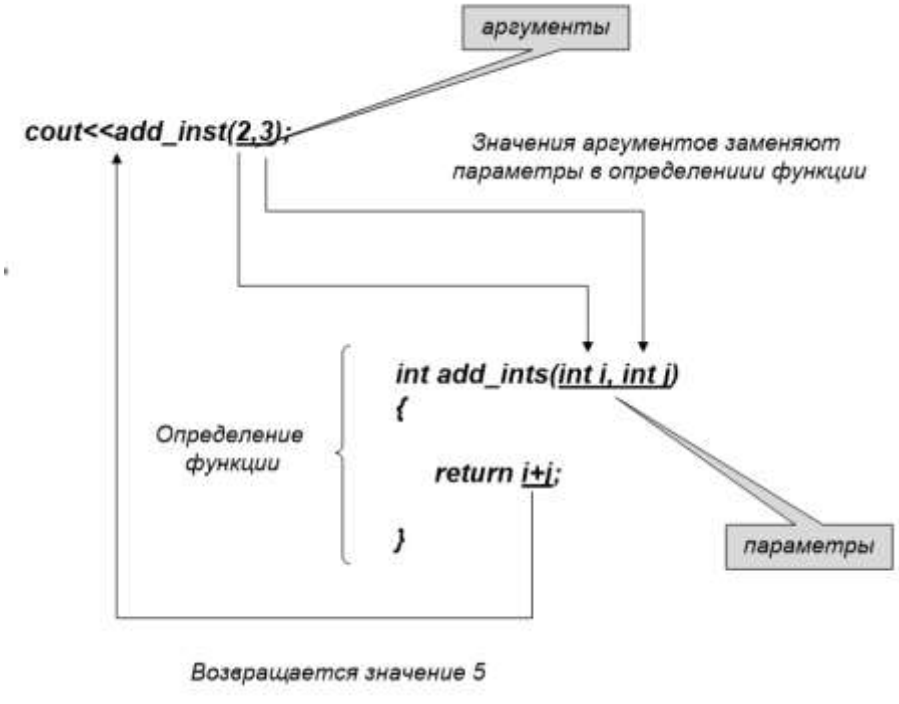
Функція це іменований блок коду, який може викликатись з інших місць програми на ім'я. Функція може приймати різні аргументи як вхідні параметри. Функція може повертати результат своєї роботи.

Одна з головних переваг, що надаються функцією, полягає в тому, що вона може бути виконана стільки разів, скільки необхідно, у різних точках програми.

Функція має ім'я, яке як ідентифікує її, так і служить для виклику на виконання всередині програми. Ім'я функції глобальне, але не обов'язково унікальне в C++, як ви побачите це у наступному розділі; однак, функції, які виконують різні дії, зазвичай повинні мати різні імена.

Імена функцій підпорядковуються тим самим правилам, як і імена змінних. Тобто ім'я функції - це послідовність літер та цифр, що починається з літери, причому знак підкреслення теж вважається літерою. Ім'я функції має зазвичай відбивати те, що вона робить.

Ви передаєте інформацію функції за допомогою аргументів, специфікованих під час її виклику. Ці аргументи повинні відповідати параметрам, що відображаються у визначенні функції. Коли функція виконується, то вказані вами аргументи замінюють параметри, використані її визначенні. Код функції виконується так, ніби він був написаний із застосуванням значень ваших аргументів. На малюнку показані відносини між аргументами під час виклику функції та параметрами, заданими у її визначенні.



Навіщо потрібні функції

Одна з головних переваг, що надаються функцією, полягає в тому, що вона може бути виконана стільки разів, скільки необхідно, у різних точках програми. Без такої можливості упаковувати блоки коду у функції, програми були б набагато більшими, оскільки тоді довелося б повторювати той самий код скрізь, де він може знадобитися. Але реальна необхідність у функціях викликана тим, щоб можна було розбивати програму на легко керовані фрагменти для незалежної розробки та тестування.

Уявіть собі справді велику програму – скажімо, у мільйон рядків коду. Програму такого розміру неможливо написати без функцій. Функції дозволяють сегментувати програму так, що її можна писати частинами, і тестувати кожну частину незалежно, перш ніж з'єднувати її з іншими частинами. Це також дає можливість розподілити роботу між членами команди розробників, де кожен член команди відповідає за чітко визначену частину програми з добре визначеним функціональним інтерфейсом для решти коду. Наприклад, одна функція може виводити масив на друк, інша, наприклад, сортувати той самий масив.

Структура функції

Як ви вже бачили, коли писали функції main(), функція складається із заголовка функції, який ідентифікує її, а за ним слідує тіло функції, укладене у фігурні дужки, де міститься її виконуваний код.

Спробуємо написати функцію, яка зводитиме значення в заданий ступінь, тобто обчислювати результат множення значення x на себе n разів.

```
// Приклад: функція для обчислення x у ступені n, де n більше або 0
double power(double x, int n)           // Заголовок функції
{ // Тіло функції починається тут...
        double result = 1.0;            // Тут зберігається результат
        for (int i = 1; i <= n; i++)
                result *= x;
        return result;
}       // ...і закінчується тут


void main(void)
{
        std::cout << power( 2, 3 );
}
```

Заголовок функції

Спочатку розглянемо у цьому прикладі заголовок функції. Наступний рядок – перший рядок функції:

double power(double x, int n) // Заголовок функції

Він складається із трьох частин, які описані нижче.

Тип значення, що повертається (в даному випадку - double).

Ім'я функції (у разі - power).

Параметри функції, укладені в дужки (в даному випадку-x і n, типу double та int відповідно).

Значення, що повертається, повертається викликає функції, тому, коли дана функція викликається, то її результат типу double підставляється у вираз, з якого вона викликана.

Наша функція має два параметри: x – значення типу double, яке потрібно звести у ступінь, та n – значення ступеня типу int. Ці змінні параметри беруть участь у обчисленнях, що виконуються функцією, разом з іншою змінною - result, оголошеною в її тілі. Імена параметрів та будь-які змінні, визначеноні в тілі функції є локальними по відношенню до неї.

Як і у випадку зі змінними, функції треба оголошувати. У цьому прикладі заголовок функції говорить про те, що далі слідує код певної функції з ім'ям power, якою будуть передані як аргументи два параметри типу double і int. Ці параметри звуться «формальних аргументів». Це зовсім не означає, що при виклику функції в неї потрібно зрадити обов'язково змінні x і n. Формальні аргументи лише говорять нам про кількість та тип параметрів.

Коли ж ми викликаємо функцію

std::cout << power(2, 3);

ми передаємо до неї вже «фактичні аргументи». Це можуть бути змінні або безпосередньо числові значення

В кінці заголовка функції і після фігурної дужки, що закриває її тіла, точка з комою не потрібна.

Загальна форма заголовка функції .

Загальна форма заголовка функції може бути записана так:

тип_повернення ім'я_функції (список_параметрів)

тип_повернення може бути будь-яким легальним типом. Якщо функція не повертає значення, тип повернення вказується ключовим словом void. Ключове слово void також застосовується для позначення відсутності параметрів, тому функція, яка не має параметрів та не повертає значення, повинна мати таку форму заголовка:

```
void my_function(void)
```

Порожній список параметрів також означає, що функція не має аргументів, тому можна пропустити ключове слово void між дужками:

```
void my_function()
```

Функція з типом повернення void не повинна використовуватися у складі виразів у програмі, що викликає.

Тіло функції

Усі необхідні обчислення функції виконуються операторами в її тілі, яке слідує за заголовком. Тіло функції нашого останнього прикладу починається з оголошення змінної result, ініціалізованої значенням 1.0. Змінна результат локальна по відношенню до функції, як і всі автоматичні змінні, оголошені в її тілі. Це означає, що змінна результат

припиняє існування після того, як функція завершить роботу. Швидше за все, у вас негайно виникне питання: якщо змінна результат припиняє існування після завершення роботи функції, то як вона може бути повернена? Відповідь у тому, що при цьому автоматично створюється копія значення, що підлягає поверненню, і копія повертається в програму.

Обчислення проводиться у циклі for. Керуюча змінна циклу і оголошена в самому циклі, і передбачається, що вона послідовно набуває значень від 1 до n. Змінна результат множиться на x при кожній ітерації циклу, тому це відбувається n разів, щоб згенерувати необхідне значення. Якщо n дорівнює 0, то операторциклу нічого очікувати виконано жодного разу, оскільки умовне вираз відразу поверне false, і result залишиться рівним 1.0.

Параметри та всі змінні, оголошені в тілі функції, локальні до неї. Ніщо не заважає використовувати ті ж імена змінних в інших функціях, для інших цілей. Справді, інакше було б надзвичайно важко забезпечити унікальність імен змінних усередині програми, що складається з багатьох функцій, особливо якщо ці функції розробляє не одна людина.

Область видимості змінних, оголошених усередині функції, визначається так само, як згадувалося. Змінна створюється в точці її оголошення та припиняє своє існування в кінці блоку, в якому була оголошена. Проте існує різновид змінних, що становить виняток із цього правила - змінні, оголошені як static.

Оператор return.

Оператор return повертає значення result у точку виклику функції. Загальна форма оператора return така:

return вираз;

де вираз має обчислюватися як значення типу, визначеного вспритність функції для повернення значення. Вираз може бути будь-яким, яке хочете, доки воно в результаті віддає значення необхідного типу. Вираз може включати виклики функцій і навіть виклик тієї самої функції, де воно з'являється.

Якщо тип повернення функції специфікований як void, то за оператором return не повинно бути жодного виразу. Воно має записуватися дуже просто:

return;

Використання функцій.

У точці, де в програмі використовується функція, компілятор повинен знати щось про неї, щоб скомпілювати її виклик. Йому потрібна достатня інформація, щоб ідентифікувати функцію, та переконатися, що ви застосовуєте її коректно. І якщо функція, яку ви маєте намір використовувати, не з'явилася десь раніше в тому ж вихідному файлі, ви повинні оголосити функцію за допомогою оператора, який називається прототипом функції.

Прототипи функцій.

Прототип надає базову інформацію, яку компілятор повинен перевірити, щоб переконатися, що функція використовується коректно. Він визначає параметри, що передаються функції, її ім'я та тип значення, що повертається. По суті, прототип містить ту саму інформацію, що міститься в заголовку функції з додаванням точки з комою. Зрозуміло, що кількість параметрів та їх типи у прототипі функції повинні бути такими, як у її заголовку.

Прототипи функцій, що викликаються з іншої функції, повинні з'являтися перед операторами, де ці функції викликаються, і тому зазвичай розміщуються на початку файлу програми. Заголовні файли, які ви включаєте для використання стандартних бібліотечних функцій, також включають прототипи цих бібліотечних функцій.

Для прикладу функції power() ви можете написати наступний прототип:

```
double power(double value, int index);
```

Не забувайте про точку з комою в кінці прототипу функції! Без цього ви отримаєте повідомлення про помилку від компілятора.

Зверніть увагу, що імена параметрів у прототипі функції можуть відрізнятись від тих, що застосовувалися в заголовку функції при її визначенні. Найчастіше в прототипах вказуються ті ж імена, що й у заголовку визначення функції, але це не є обов'язком.тельно має бути так.

Ви можете використовувати більш довгі і виразні імена параметрів у прототипі функції, щоб пояснити їх призначення, а потім вказати більш короткі імена тих самих параметрів у визначенні функції, де довгі імена могли б занапастити код і зробити його менш читабельним.

За бажання ви можете взагалі пропустити імена параметрів у прототипі функції та просто написати так:

```
double power(double, int);
```

Це надає достатньо інформації компілятор, щоб він виконав свою роботу; проте краще використовувати деякі осмислені імена в прототипе, тому що це підвищує читабельність і в деяких випадках взагалі відрізняє ясний код від заплутаного. Якщо у вас є функція з двома параметрами одного і того ж типу (припустимо, наприклад, що у нас index у функції power () також був би типу double), то вибір для них осмислених імен показує, який параметр іде першим, а який другим.

//Приклад: оголошення, визначення та застосування функції.

```
#include <iostream>
double power(double x, int n);          //Прототип функції
using namespace std;
void main()
{       int index = 3;
        double x = 3.0;
        double y=0.0;
        y = power (5.0, 3);             //Передача констант як аргументів
        cout << endl << " 5.0 v kube = " << y;
        cout << endl << " 3.0 v kube = "
                << power(3.0, index); //Виведення поверненого значення
        x = power(x, power(2.0, 2.0));          //застосування функції
        cout << endl << "x = " << x;
        cout << endl;
}
```

```
//Функція для обчислення x у ступені n, де n більше або 0
double power(double x, int n)          // Заголовок функції
{ //Тіло функції починається тут…
        double result = 1.0;
        for (int i = 1; i <= n; i++)
                result *= x;
        return result;
}//… і закінчується тут
```

```
 5.0 v kube = 125
 3.0 v kube = 27
x = 81
```

Після звичайного оператора #include для підтримки введення-виведення йде прототип функції power(). Якщо ви видалите його і спробуєте перекомпілювати програму, компілятор не зможе обробити виклики функції в main (), а видасть натомість серію повідомлень про помилки.

У цьому прикладі використано ключове слово void у функції main (), де зазвичай з'являється список параметрів, щоб показати, що параметри не використовуються. Ключове слово void також може застосовуватися як тип повернення функції, щоб показати, що функція не повертає значення. Якщо ви визначаєте тип повернення як void, то не повинні поміщати жодного значення поруч із оператором return всередині функції; інакше ви отримаєте від компілятора повідомлення про помилку.

Щоб використовувати функцію power () для обчислення 5,03 і зберегти результат у змінній у нашому прикладі використовується наступний оператор:

```
y = power (5.0, 3);
```

Тут значення 5.0 та 3 – аргументи функції. В даному випадку вони є константами, але ви можете використовувати будь-які вирази як аргументи, якщо вони дають результат потрібного типу.

Аргументи функції power() підставляються замість параметрів x та n, які використовуються у визначенні функції. Обчислення проводиться із застосуванням цих значень, а копія результату, 125, повертається викликає функції main (), де присвоюється змінної y. Ви можете думати про функцію як значення в операторі або виразі, де вона з'являється.

Потім у прикладі значення змінної виводиться на екран:

```
cout << endl<< " 5.0 v kube = " <<y;
```

Далі виклик функції застосовується у складі оператора вивода:

```
cout <<endl
        << " 3.0 v kube = "
        <<power(3.0, index); //Виведення поверненого значення
```

Тут значення, яке повертається функцією, передається безпосередньо у вихідний потік. Оскільки ви ніде не зберігаєте це значення, воно в інший спосіб вам недоступне. Перший аргумент у цьому виклику функції – константа, а другий – змінна.

Після цього функція power() використовується ще раз в операторі:

x = power(x, power(2.0, 2.0)); // Використання функції як аргументу

І тут функція power () викликається двічі. Перший виклик - правий у виразі, і його результат є другим аргументом для другого, лівого виклику. Хоча обидва аргументи в подвыражении power (2 . 0, 2.0) є літералами типу double, функція насправді викликається з першим аргументом 2 . 0 і другим - цілим літералом 2. Компілятор перетворює значення double, наведене як другий аргумент, до типу int, оскільки знає на підставі прототипу, що типом другого аргументу повинен бути int:

double power(double x, int n); // Прототип функції

Результат 4.0 типу double повертається першим викликом функції power() після перетворення типу int значення 4 передається як другий аргумент наступному виклику функції, де перший аргумент - x. Оскільки x має значення 3.0 значення обчислюється 3,04 і результат, рівний 81.0, присвоюється x.

Цей оператор містить у собі два неявні перетворення типу double у тип int, вставку яких забезпечує компілятор. При такому перетворенні даних можлива втрата даних, тому компілятор видає попереджувальні повідомлення, коли таке трапляється, хоча він сам і вставив це перетворення. Зазвичай покладатися на автоматичне перетворення типів, потенційно загрожує втратою даних - небезпечна практика у програмуванні, і зовсім не випливає з коду, що таке перетворення було навмисним. Набагато краще, коли необхідно, явно вказувати в коді перетворення типу, використовуючи для цього операцію static_cast. Тобто останній оператор у цьому прикладі краще переписати так:

x = power (x, static_cast <int> (power (2.0, 2))));

Таке кодування оператора дозволяє уникнути обох попереджень компілятора, що викликає вихідна версія. Застосування статичного приведення не виключає можливості втрати даних під час перетворення одного типу на інший. Але оскільки воно зазначено явно, то компілятор ясно, що це входить у ваші наміри.

Передача аргументів на функцію.

Аргументи, які ви визначаєте при виклику функції, зазвичай повинні відповідати за типом та послідовністю параметрам, заданим у її визначенніленні. Як ви бачили з останнього прикладу, якщо тип аргументу, вказаного під час виклику функції, не відповідає типу параметра, зазначеному в її визначенні, то там, де можливо, відбувається перетворення до необхідного типу відповідно до правил приведення операндів, описаних раніше. Якщо таке приведення неможливе, ви отримуєте від компілятора повідомлення про помилку. Однак навіть якщо приведення можливе і код компілюється, це може призвести до втрати даних (наприклад, коли тип long наводиться до short), тому його слід уникати.

Існує 3 основних способи передачі аргументів у функцію:

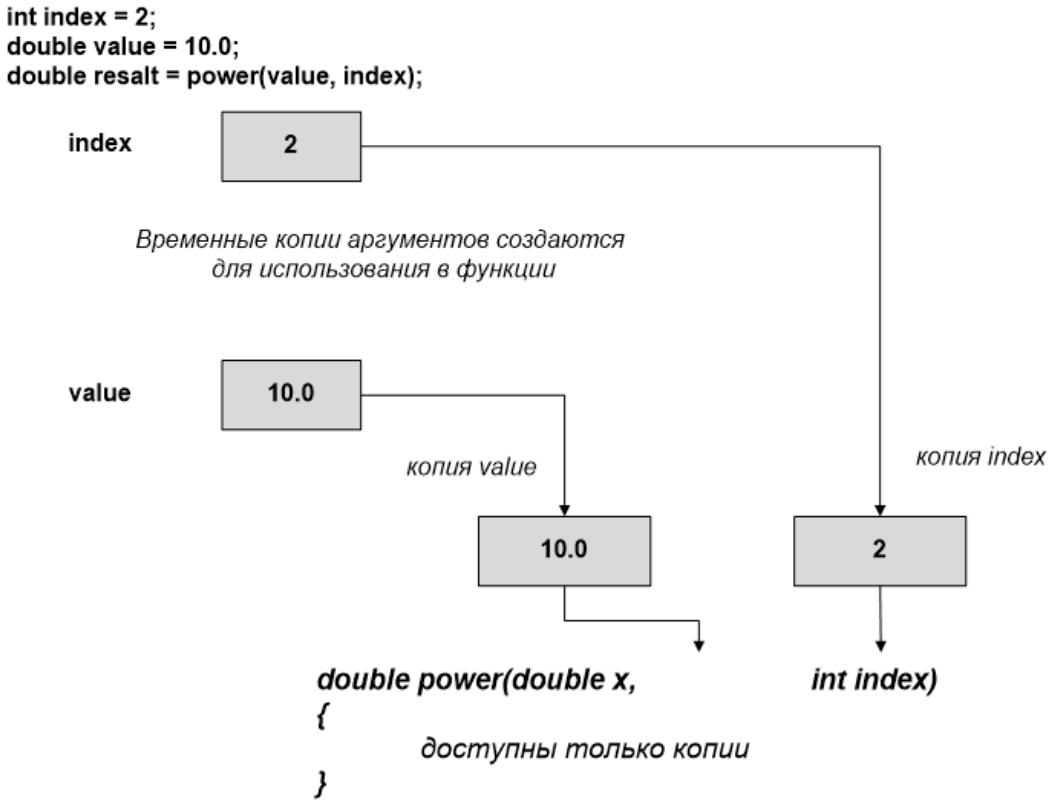
- передача за значенням;
- передача за посиланням;
- передача на адресу.

За умовчанням, аргументи C++ передаються за значенням. Коли аргумент передається за значенням, його значення копіюється в параметр функції.

Механізм передачі за значенням.

Коли застосовується такий механізм, то змінні чи константи, які ви специфікуєте як аргументи, взагалі не передаються у функцію. Натомість створюються копії аргументів, і ці копії використовуються як передані значення. На малюнку показана діаграма стосовно функції power().

Кожного разу, коли функція power () викликається, компілятор створює копії переданих аргументів і розміщує їх у тимчасовій ділянці пам'яті. Під час виконання функції всі посилання на її параметри відображаються на ці копії аргументів.



Один із наслідків механізму передачі за значенням у тому, що функція неспроможна безпосередньо модифікувати аргументи.

```
// Марна спроба модифікації аргументів коду, що викликає.
#include <iostream>
int incr10(int num);              // Прототип функції
using namespace std;
int main(void)
{       int num = 3;
        cout << endl
                << "incr10(num) = " << incr10(num)
                << endl
                <<"num=" <<num<<endl;
}
```

```
// Функція збільшення змінної на 10
int incr10(int num)
{       num + = 10;             // Спроба змінити аргумент
        return num;             // Повернення зміненого значення
}
```

Звичайно, ця програма приречена на невдачу. Якщо ви запустите її, то отримаєте наступний висновок:

```
incr10(num) = 13
num = 3
```

Опис отриманих результатів.

Висновок підтверджує, що вихідне значення num залишається незачепленим. Зміна піддається копію num, яка була створена і потім відкинута при виході з функції.

Зрозуміло, що механізм передачі за значенням забезпечує високий рівень захисту аргументів виклику від пошкодження недобросовісною функцією, але може статися, що ви дійсно хочете, щоб передані аргументи були модифіковані функцією.

Передача аргументів на адресу - це передача адреси змінної-аргументу (а не вихідної змінної). Оскільки аргумент є адресою, то параметром функції має бутипокажчик. Потім функція може розіменувати цей покажчик для доступу або зміни вихідного значення.

Покажчики як аргументи функцій.

Коли ви використовуєте вказівники як аргументи, механізм передачі за значенням працює, як і раніше; однак вказівник - це адреса іншої змінної, і якщо ви візьмете копію цієї адреси, то вона як і раніше вказуватиме на ту саму змінну. Таким чином, застосування вказівника як параметр дозволяє вашій функції отримати сам аргумент, а не його копію.

Для демонстрації ефекту можна змінити останній приклад, щоб використати покажчик:

```
//Успішна спроба модифікувати аргумент
#include <iostream>
int incr10(int * pnum);                 // Прототип функції
using namespace std;
int main(void)
{       int num = 3;
        int* pnum = #
        cout << endl
                << "adress has been received_1_ = " << pnum<< endl;
                cout << "incr10(num) = " << incr10(&num);
                        cout<< endl<< "num = " << num<<endl;
        return 0;
}
```

```
// Функція збільшення змінної на 10
int incr10(int*pnum)  // Застосування такого імені може допомогти...
{       cout<< "adress has been received_2_ = " << pnum << endl;
        * pnum + = 10;                  // Спроба змінити аргумент
        cout <<" new num = "<< *pnum << endl;
        return *pnum;// Повернення зміненого значення
}
```

Результат виконання програми

```
adress has been received_1_ = 0113F76C
adress has been received_2_ = 0113F76C
 new num = 13
incr10(num) = 13
num = 13
```

Точне значення адреси на вашому комп'ютері може відрізнятись від показаного тут, але два відображені значення адреси мають бути ідентичними.

Опис отриманих результатів.

Принципова відмінність цього прикладу від попередньої версії полягає у передачі покажчика pnum замість вихідної змінної num. Прототип функції тепер специфікує тип параметра як покажчик на int, а функції main () оголошений покажчик pnum, ініціалізований адресою num.

Функція main () і функція incr10 () виводять відповідно відправлену та прийняту адресу, щоб показати, що в обох місцях використовується та сама адреса.

Результат програми показує, що цього разу значення num було змінено і має значення, ідентичне тому, що повернула функція.

У зміненій версії функції incr10 () тепер і оператор, який змінює передане значення, і оператор return виконують розіменування вказівника, щоб звернутися до значення, що зберігається в ньому.

У цьому прикладі є певна доза лукавства. Ми дізналися, як передавати в функцію покажчик, як можна за допомогою передачі покажчика модифікувати змінну, але аргумент, як і раніше, не був змінений. В даному випадку аргументом був вказівник і він не змінився. Змінилася лише змінна, яку він вказував. Як дійсно змінити аргумент у функції, ми дізнаємося трохи пізніше.

Використання посилань.

Посилання в C++ - це альтернативне ім'я об'єкта.

Оголошення посилань дуже схоже на оголошення покажчиків, тільки замість зірочки* пишеться амперсанд &. При оголошенні посилання має бути ініціалізована.

Що таке посилання?

Посилання впрограмування- це об'єкт, що вказує на певні дані, але не зберігає їх. Отримання об'єкта за посиланням називається розйменуванням.

Посилання не євказівникома просто є іншим ім'ям для об'єкта. Головна відмінність посилання від покажчиків у цьому, що покажчик -це ціле число і тому йому доступні

операції з цілими числами, а посилання доступні лише операції копіювання і розіменування.

У мовах програмування посилання може бути реалізована як змінна, що містить адресу осередки пам'яті.

Посилання - цепсевдонім для іншої змінної. Вона має ім'я, яке можна використовувати замість вихідного імені змінної. Оскільки це псевдонім, а не вказівник, змінна, для якої вона визначена, повинна бути вказана, коли оголошується посилання, і на відміну від покажчика, посилання не може бути змінено, щоб представляти іншу змінну.

Оголошення та ініціалізація посилань

Припустимо, ви оголосили змінну наступним чином:

```
long number = 0;
```

Після цього ви можете оголосити посилання на цю змінну за допомогою такого оператора оголошення:

```
long& rnumber = number; // Оголошення посилання на змінну number
```

Знак амперсанд, що іде за ім'ям типу long і попередній імені змінної rnumber, говорить про те, що це оголошення посилання, а ім'я змінної, яку вона представляє - number - визначено як значення, що ініціалізує, наступне за знаком рівності; таким чином, змінна rnumber має тип "посилання на long". Після цього можна використовувати посилання замість імені вихідної змінної.

Наприклад, оператор:

```
rnumber += 10;
```

Як ефект дає збільшення значення змінної number на 10.

Порівняйте посилання rnumber із вказівником pnumber, оголошеним в операторі:

```
long* pnumber = &number; // Ініціалізація вказівника адресою
```

Тут оголошуєтьсявказівник pnumber, який ініціалізується адресою змінної number. Це дозволяє збільшувати значення number оператором на зразок:

```
* p n u m b e r  + = 1 0 ;        // Інкремент number через вказівник
```

Є суттєва різниця між використанням покажчика та використанням посилання. Покажчик повинен бути розіменований, і адреса, яку він містить, служить для доступу до змінної, що бере участь у вираженні.

У разі посилання немає потреби у розіменуванні. У певному сенсі посилання подібне до покажчика, який вже розіменований, хоча його і не можна змусити посилатися на іншу змінну. Посилання повністю еквівалентне змінній, яку вона посилається. Посилання може здатися просто альтернативною нотацією для цієї змінної, і тут вона безперечно веде себе саме таким чином. Однак коли ми говоритимемо про функції C++, ви переконаєтеся, що це не зовсім так, і що цей засіб пропонує деякі дуже вражаючі додаткові можливості

Посилання C++ дозволяє створити псевдонім (або друге ім'я) для змінних у програмі. Для оголошення посилання всередині програми вкажіть знак амперсанди (&) безпосередньо після типу параметра. Оголошуючи посилання, ви повинні відразу ж присвоїти їй змінну, для якої це посилання буде псевдонімом, як показано нижче:

```
int&alias_name=variable; // Оголошення посилання
```

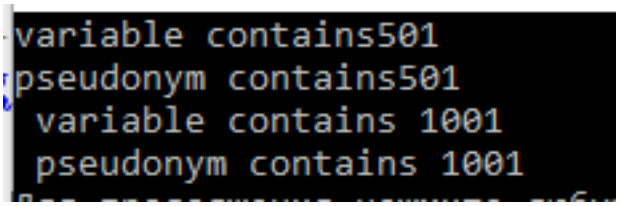
Після оголошення посилання ваша програма може використовувати або змінну , або посилання:

```
alias_name = 1001;
variable = 1001;
```

Наступна програма створює посилання з ім'ям alias_name і надає псевдоніму змінну number. Далі програма використовує як посилання, так і змінне:

```
// Приклад
#include <iostream>
using namespace std;
void main(void)
{       int number = 501;
        int &alias_name=number;              // Створити посилання
        cout<< "Variable contains" <<number<< endl;
        cout<< "pseudonym contains" <<alias_name<< endl;
        alias_name=alias_name+500;
        cout<< " variable contains " <<number<< endl;
        cout<< " pseudonym contains " <<alias_name<< endl;
}
```

Результат виконання програми



Як бачите, програма додає 500 до посилання alias_name. У результаті програма додає 500 також до відповідної змінної number, для якої посилання служить псевдонімом або другим ім'ям.

У випадку використання посилання таким чином, як щойно було показано, створює труднощі для розуміння. Однак, ви побачите, що використання посилань значно спрощує процес зміни параметрів усередині функції.

Посилання як аргументи функції.

Тепер перейдемо до другого із двох механізмів передачі аргументів у функцію. Визначення параметра функції як посилання змінює метод передачі параметра. Цей метод не передає аргумент за значенням, коли він копіюється перед тим, як бути переданим у функцію, а передається за посиланням - тобто параметр виступає псевдонімом аргументу, що передається. Це виключає будь-яке копіювання і забезпечує прямий доступ до аргументу. Це також означає, що немає необхідності в розіменуванні, яке необхідне під час передачі покажчиків на значення.

Для оголошення та ініціалізації посилання всередині програми оголосіть змінну, розміщуючи амперсанд (&) відразу після типу змінної, а потім використовуйте оператор присвоювання для призначення псевдоніма, наприклад

```
int&alias_name=variable;
```

– Ваші програми можуть передавати посилання на функцію як аргументи, а функція, у свою чергу, може змінювати відповідне значення аргументу, не використовуючи покажчиків.

– Всередині функції слід оголосити параметр як посилання, розміщуючи амперсанд (&) після типу параметра, потім можна змінювати значення параметра всередині функції без вказівників.

```
// Приклад: використання посилання модифікації аргументу
#include <iostream>
using namespace std;
intincr10(int &num);            // Прототип функції
intmain(void)
{          int num = 3;
           int value = 6;
          cout<<endl<<"incr10(num) = " << incr10(num);
          cout<<endl<< "num=" <<num;
          cout<<endl<<"incr10 (value) = " <<incr10 (value);
          cout<<endl<< "value=" << value;
          cout<< endl;
 return 0;
}
// Функція збільшення змінної на 10
intincr10(int &num)              // Функція з аргументом – посиланням
{          cout<< endl <<"result = " << num;
          num+= 10;      // Збільшити значення аргументу
           return num;    // Повернути збільшене значення
}
```

Результат виконання програми:

```
result = 3
incr10(num) = 13
num = 13
result = 6
incr10 (value) = 16
value = 16
```

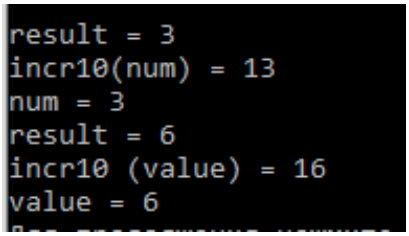
Використання модифікатора const

Щоб повідомити компілятору, що ви не збираєтесь жодним чином модифікувати параметр функції, ви можете застосувати модифікатор const. Це змусить компілятор перевірити код функції на предмет того, чи він не змінює значення аргументу, і якщо ні, то жодних повідомлень про помилки не видається при використанні константного аргументу.

Щоб побачити, як модифікатор const змінює ситуацію, модифікуємо попередню програму.

```
//Приклад: використання константного посилання.
#include <iostream>
using namespace std;
intincr10(const int&num);              // Прототип функції
intmain(void)
{         int num = 3;
          int value = 6;
          cout<<endl<<"incr10(num) = " << incr10(num);
          cout<<endl<< "num=" <<num;
          cout<<endl<<"incr10 (value) = " <<incr10 (value);
          cout<<endl<< "value=" << value;
          cout<< endl;
 return 0;
}
// Функція збільшення змінної на 10
intincr10(const int&num)               // Функція з аргументом – посиланням
{         cout<<endl <<"result = " << num;
          return num+10;         // Повернути збільшене значення
}
```

Результат виконання програми



Опис отриманих результатів.

Оголосимо змінну num в main() як const, щоб показати, що коли параметр функції incr10 () оголошений як const, то компілятор більше не видаєспілкування про помилку під час передачі константного об'єкта.

Необхідно також закоментувати оператор, який збільшує значення **num** у функції **incr10** (). Якщо ви приберете коментар з цього рядка, програма перестане компілюватись, тому що компілятор не допустить появи num у лівій частині привласнення. Коли ви визначаєте num як const у заголовку та в прототипі функції, це означає, що ви зобов'язуєтеся не модифікувати його, і компілятор перевіряє, як ви тримаєте слово.

Все працює як раніше, крім того, що змінні з main () більше не змінюються у функції.

Використовуючи аргументи-посилання, ви берете найкраще із двох світів. З одного боку, ви можете написати функцію, яка отримує прямий доступ до вихідного аргументу коду, що дозволить, дозволивши уникнути копіювання, яке передбачає механізм передачі за значенням. З іншого боку, коли ви не маєте наміру змінювати аргумент, ви отримуєте повний захист від ненавмисної зміни, використовуючи модифікатор const з посиланням.

13. ПЕРЕДАЧА МАСИВІВ НА ФУНКЦІЮ.

Коли масив використовується як аргумент функції, передається лише адреса масиву, а не копія всього масиву. При виклику функції з ім'ям масиву передається вказівник на перший елемент масиву. (Треба пам'ятати, що в C + + імена масивів без індексу - це покажчики на перший елемент масиву.)

Тому наступні оголошення функції будуть по суті рівноцінні:

```
1       void print(int numbers[]);
2       void print(int *numbers);
```

При цьому потрібно вказати тип елемента масиву.

```
#include <iostream>
using namespace std;
void show_average(double array[10]);        // Прототип функції
intmain(void)
{
        double values[10] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};

        show_average(values);        cout << endl;
        return 0;
}
// Функція висновку
void show_average(double array[10])
{       for (int i = 0; i < 10; i++)
                cout<<'\t'<< array[i];
}
```

Розмір масиву в функцію автоматично не передається, тому якщо розмір масиву заздалегідь (на етапі компіляції) не обумовлений, то потрібно передати параметр, який містить кількість елементів масиву, наприклад number_of_elements:

```
void some_function(int array[], int number_of_elements);
```

```
#include <iostream>
using namespace std;
void show_average(double array[],int n);    // Прототип функції
intmain(void)
{
        double values[10] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
        int n=10;
        show_average(values, n);      cout << endl;
        return 0;
```

```
}
// Функція висновку
void show_average(double array[],int n)
{       for (int i = 0; i<n; i++)
                cout<<'\t'<< array[i];
}
```

У функцію можна передавати масиви, але в цьому випадку масив не копіюється, навіть незважаючи на те, що при цьому застосовується передача аргументу за значенням. Ім'я масиву перетворюється на покажчик, і копія покажчика початку масиву передається у функцію за значенням.

Це досить вигідно, бокопіювання великих масивів потребує значних витрат часу. Однак, у цьому випадку елементи масиву можуть бути змінені всередині функції, і тому масив - єдиний тип, який не може бути переданий за значенням.

Плюси та мінуси цього можна проілюструвати на прикладі функції, що обчислює середню величину значень, переданих у масиві.

```
// Приклад: передача масиву на функцію
#include <iostream>
using namespace std;
doubleaverage(double array[],int count);     // Прототип функції
intmain(void)
{       doublevalues[] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,10.0};
        cout << endl
                << "AV="
                <<average(values, (sizeofvalues) / (sizeofvalues[0]));
        cout << endl;
        return 0;
}
// Функція для обчислення середнього
doubleaverage(double array[],int count)
{       doublesum = 0.0;                    // Тут накопичується сума
        for(inti = 0; i <count; i++)
                sum +=array[i];             // Підсумовувати елементи масиву
        returnsum /count;                   // Повернути середнє
}
```

Результат виконання програми:

Середнє = 5.5

Опис отриманих результатів.

Функція average() призначена для роботи з масивами будь-якої довжини. Як видно з її прототипу, вона приймає два аргументи: масив та лічильник кількості елементів.

Оскільки хочемо, щоб вона працювала з масивами довільної довжини, параметр масиву вказаний без специфікації виміру. Функція викликається в main () наступним оператором:

```
cout << endl
<< "AV="
<< average(values, (sizeof values)/(sizeof values[0]));
```

Першим аргументом передається ім'я масиву,values, а другим - вираз, який обчислюється як кількість елементів масиву.

Операція sizeof повертає ціле значення, яке означає кількість байт, що займає її операнд. Застосовується до масиву та елементу масиву.

Усередині тіла функції обчислення виконується так, як можна очікувати.

Висновок програми підтверджує, що все працює як треба.

Використання нотації покажчиків під час передачі масивів

Ім'я масиву може передаватися як покажчика (точніше, як копії покажчика), тому всередині функції ви взагалі зобов'язані працювати з даними як із масивом. Можна так змінити функцію в прикладі, щоб всюди використовувати нотацію вказівників, незважаючи на той факт, що йдеться про масив.

```
// Приховування масиву функції покажчиком
#include <iostream>
using namespace std;
double average(double* array, int count);    // Прототип функції
int main(void)
{       double values[] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,10.0};
        cout << endl
                << "AV="
                <<average(values, (sizeofvalues) / (sizeofvalues[0]));
        cout << endl;
        return 0;
}
// Функція для обчислення середнього
doubleaverage(double* array,int count)
{       doublesum = 0.0;                    // Тут накопичується сума
        for(inti = 0; i <count; i++)
                sum +=array[i];             // Підсумовувати елементи масиву
        returnsum /count;                   // Повернути середнє
}
```

Результат виконання програми:

```
Середнє = 5.5
```

Висновок програми виглядає так само, як у попередньому прикладі.

Опис отриманих результатів.

Як бачите, для того, щоб програма працювала з масивом як із покажчиком, до неї потрібно внести зовсім небагато змін. Змінюється прототип і заголовок функції, хоча ці зміни є абсолютно необхідними. Якщо повернути їх до вихідного вигляду, де перший параметр специфіковано як масив double, і залишити тіло функції в термінах покажчика, вона буде працювати так само добре. Найцікавіший аспект цієї версії тіла функції криється в операторі циклу for:

```
sum += *array++;      // Підсумовувати елементи масиву
```

Можна подумати, що тут порушується правило про те, що не можна модифікувати адресу, специфіковану як ім'я масиву, оскільки збільшується адреса, що зберігається в array. Але насправді це правило не порушується. Механізм передачі за значенням створює копію вихідної адреси масиву і передає його, тому в тілі функції модифікується лише ця копія (початкова адреса масиву залишається незмінною).

В результаті щоразу, коли ви передаєте одномірний масив у функцію, то вільні трактувати передане значення в будь-якому сенсі як покажчик, а також довільним чином змінювати адресу, що зберігається в ньому.

Передача у функцію багатовимірних масивів

Передача на функцію багатовимірного масиву досить проста. Наступний оператор оголошує двовимірний масив beans:

```
double beans[2][4];
```

Ви можете написати прототип гіпотетичної функції field () приблизно так:

```
double field(double beans [2] [4]);
```

Вас може здивувати, як компілятор може знати, що це визначення масиву з розмірністю, вказаною в квадратних дужках, а не окремого елемента масиву?

Відповідь проста - ви не можете написати окремий елемент масиву як параметр у визначенні або прототипі функції, хоча можете передати його як аргумент під час її виклику. Параметр, який приймає окремий елемент масиву як аргумент, повинен бути оголошений як окрема змінна. Контекст масиву тут не застосовується.

При визначенні багатовимірного масиву як параметр можна також пропустити перший вимір масиву. Звичайно, функція повинна якось дізнатися про розмір першого виміру. Наприклад, ви можете написати так:

```
double field(double beans[] [4], int index);
```

Тут другий параметр може передати потрібну інформацію щодо величини першого виміру. У цьому випадку функція може оперувати двовимірним масивом зі значенням першого виміру, зазначеним другим аргументом функції, і фіксованим значенням другого виміру, що дорівнює 4.

Використання подібної функції показано в наведеному нижче прикладі.

```
// Передача на функцію двовимірного масиву

#include <iostream>

using namespace std;

doublefield(double array[][4],int n);

intmain(void)

{       doublebeans[3][4] = { { 1.0, 2.0, 3.0, 4.0 },
                              { 5.0, 6.0, 7.0, 8.0},
```

```
                                        {9.0, 10.0, 11.0, 12.0}};
        cout << endl<< " harvest = " <<field(beans,sizeofbeans /sizeofbeans[0]);
        cout << endl;
        return 0;
}
// Функція для обчислення всього обсягу врожаю
doublefield(double beans[][4],int count)
{       double sum = 0.0;
        for(inti = 0; i <count; i++)              // Цикл за рядками
                for(intj = 0; j<4; j++)          // Цикл за елементами рядка
                        sum +=beans[i][j];
        returnsum;
}
```

Результат виконання програми:

harvest =78

Опис отриманих результатів.

Були використані різні імена параметрів у прототипі та заголовку функції.

Перший параметр визначений як масив з довільною кількістю рядків, кожен з яких містить чотири елементи. Функція викликається з масивомbeans, що складається з трьох рядків.

Другий аргумент визначено як приватне від розподілу загального розміру масиву в байтах розмір його першого рядка. В результаті обчислення це дає кількість рядків у масиві.

Вся обчислювальна робота функції виконується у вкладених циклах for, де внутрішній цикл підсумовує елементи окремого рядка, а зовнішній цикл повторюється кожного рядка.

//Приклад: дано натуральне число n, дійсна матриця n×6. Знайти середнє арифметичне кожного зі стовпців.

```
#include <iostream>
#include<iomanip>
#include <stdlib.h>
#include <time.h>
//Виділення пам'яті
void reservMemory(int m,int n,double**& matrix);
//Заповнення масиву випадковими числами
void fillRandomMatrix(int m,int n,double** matrix);
//Знайти середнього арифметичного кожного зі стовпців
void findAVG(int m,int n,double** matrix);
//Звільнення пам'яті
void freeMemory(int m,int n,double** matrix);
```

```
int main(void)
{       srand(time(NULL));
        Int m, n, i, j;             //індекси масивів
        double** matrix;            //покажчик на динамічний масив
        matrix = 0;
        m = 6;
        setlocale(LC_ALL,"Український");
        std::cout<< "Введіть розмір n: " << "\n";
        std::cin >> n;              //Введення індексу масиву
        reservMemory(m, n, matrix);
        fillRandomMatrix(m, n, matrix);
        findAVG(m, n, matrix);
        freeMemory(m, n, matrix);
        return 0;
}
//Виділення пам'яті
// У разі використовується посилання подвійний покажчик саме оскількив функції вдається
//копія аргументу. Оскільки ми збираємося динамічно виділяти пам'ять, ця копія призведе
//до втрати взаємозв'язку між покажчиком matrix функції main() та її копією функції
//reservMemory().
void reservMemory(int m,int n,double** &matrix)
{       int i;
        matrix=new double* [m];
        for(i = 0; i <m; i++)
        {       matrix[i] =new double[n];
        }
}
 //Заповнення масиву випадковими числами
void fillRandomMatrix(int m,int n,double** matrix)
{       inti, j;
        for(i = 0; i <m; i++)
        {       for(j = 0; j <n; j++)
                {       matrix[i][j] =rand() % 100 - 50;
                        std::cout <<std::setw(10)<< matrix[i][j]<< ";
                }
                std::cout<< '\n';
        }
```

```
}
//Знаходження середнього арифметичного кожного зі стовпців.
void findAVG(int m,int n,double** matrix)
{	inti, j;
	doubles=0;
	for(j = 0; j <n; j++)
	{	for(i = 0; i <m; i++)
		{	s +=matrix[i][j];	}
		std::cout << "j=" << j << " avg : " << s/ m <<std:: endl;
	}
}
//Звільнення пам'яті
void freeMemory(int m,int n,double** matrix)
{	int i;
	for(i = 0; i <m; i++)
	{
		delete[] matrix[i];
	}
	delete[] matrix;
}
```

Результат виконання програми:

```
Введите размер n:
6
        24         -7         23         41         42         27
        26        -49         26          9        -43         14
       -36          2         28         -8         20         45
        41        -47        -43        -30        -28        -32
       -33        -21        -46         39          9         13
        19        -12        -36        -17         12        -42
j= 0 avg : 6.83333
j= 1 avg : -15.5
j= 2 avg : -23.5
j= 3 avg : -17.8333
j= 4 avg : -15.8333
j= 5 avg : -11.6667
Для продолжения нажмите любую клавишу . . .
```