МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Л. М. Лутай, А. О. Нікітін, В. М. Бурдейна

СТВОРЕННЯ ДОДАТКІВ ДЛЯ ПРОГРАМНО-АПАРАТНИХ КОМПЛЕКСІВ

Конспект лекцій

Харків
«Факт»
2025

УДК 004.78:004.43
Л 86

Рецензенти: канд. техн. наук, доц. С. Ю. Маренич,
канд. техн. наук, доц. Д. О. Пшеничников

Лутай Л. М., Нікітін А. О., Бурдейна В. М.

Л 86 Створення додатків для програмно-апаратних комплексів [Електронний ресурс] : конспект лекцій. Харків : «Факт», 2025. 105 с.

Розглянуто основні принципи та синтаксичні конструкції щодо створення додатків на мові програмування С++ для програмно-апаратних комплексів.

Для здобувачів освіти першого та другого курсів, котрі навчаються за освітньо-професійною програмою «Комп'ютерно-інтегровані технологічні процеси і виробництва» за спеціальністю «174 Автоматизація, комп'ютерно-інтегровані технології та робототехніка». Конспект лекцій буде також корисним здобувачам освіти із спеціальностей «141 Електроенергетика, електротехніка та електромеханіка» та «175 Інформаційно-вимірювальні технології».

УДК 004.78:004.43

Іл. 10. Табл. 8. Бібліогр.: 6 назв

© Лутай Л. М., Нікітін А.О., Бурдейна В. М., 2025

ВСТУП

Одним із складових програмно-апаратних комплексів виступає мікроконтролер. Мікроконтролери – це невеликі обчислювальні пристрої, які інтегрують процесор, пам'ять і периферійні пристрої на одному чипі. Вони широко використовуються в різних галузях, таких як побутова електроніка, автомобільна промисловість, медичні пристрої та системи автоматизації [1]. Основна перевага мікроконтролерів полягає у їх здатності виконувати специфічні завдання з мінімальними витратами енергії та ресурсів.

Мікроконтролери можна зустріти практично в будь-якому технічному виробі, в якому потрібно вирішувати завдання вимірювання, обробки інформації та управління. Це може бути електропобутова техніка, пральні машини, мікрохвильові печі, вимірювальні прилади або засоби комунікації, забезпечуючи їх ефективну та безперебійну роботу.

Мікроконтролери часто застосовують у пристроях, де потрібно виконання простих, але критично важливих завдань [2]. В автомобільній промисловості мікроконтролери використовуються для керування системами двигуна, антиблокувальною системою гальм та іншими важливими функціями, забезпечуючи безпеку й ефективність автомобілів.

Мікроконтролери знаходять широке застосування в медичних пристроях, таких як глюкометри та кардіостимулятори, де вони забезпечують точне та надійне виконання медичних процедур. Для програмування мікроконтролерів часто застосовуються мова програмування C++ та її діалекти [3].

Лекція 1. ОСНОВИ ПРОГРАМУВАННЯ НА С++

Мова програмування С++ являє собою високорівневу компільовану мову програмування загального призначення зі статичною типізацією, яка підходить для створення різних додатків. На сьогодні С++ є однією із найпопулярніших і найпоширеніших мов.

Алфавіт мови

Програму мовою С++ можна розчленувати на такі найпростіші елементи (лексеми):

- ідентифікатори (символічні імена): великі та малі літери англійського алфавіту, цифри 0..9;

Ідентифікатор у С++ – це послідовність із букв англійського алфавіту, цифр і символів підкреслення, що починається не з цифри. При складанні ідентифікаторів треба дотримуватись двох правил:

1. Великі та малі літери різняться, тому Summa і summa є різні ідентифікатори.

2. Число символів в ідентифікаторі може бути практично будь-яким.

Ідентифікатори: змінні, константи, імена функцій, імена класів і т. д.

- константи (їх значення встановлюється один раз і згодом не можна її змінити. На початку визначення константи пишеться ключове слово const);
- операції (=, -, +, /, * і т. д.);
- роздільники, у тому числі коментарі (символ пробілу, символ табуляції, символ повернення каретки, крапка з комою, коментарі, всі знаки операцій).

С++ для зберігання даних використовуються змінні. Змінна має тип, ім'я та значення. Тип визначає, яку інформацію може зберігати змінна.

Перед використанням будь-яку змінну слід визначити. Найпростіше визначення змінної:

int age;.

У попередньому коді визначено змінну age, яка має тип int. Оскільки визначення змінної є інструкцією, то після нього ставиться крапка з комою.

Кожна змінна має певний тип. І цей тип визначає, які значення може мати змінна, які операції з нею можна робити і скільки байтів у пам'яті вона займатиме.

Відмінною рисою змінних є те, що можна багаторазово протягом роботи програми змінювати їх значення.

Класифікація типів даних

На рис. 1 наведено схематично класифікацію типів даних С++.

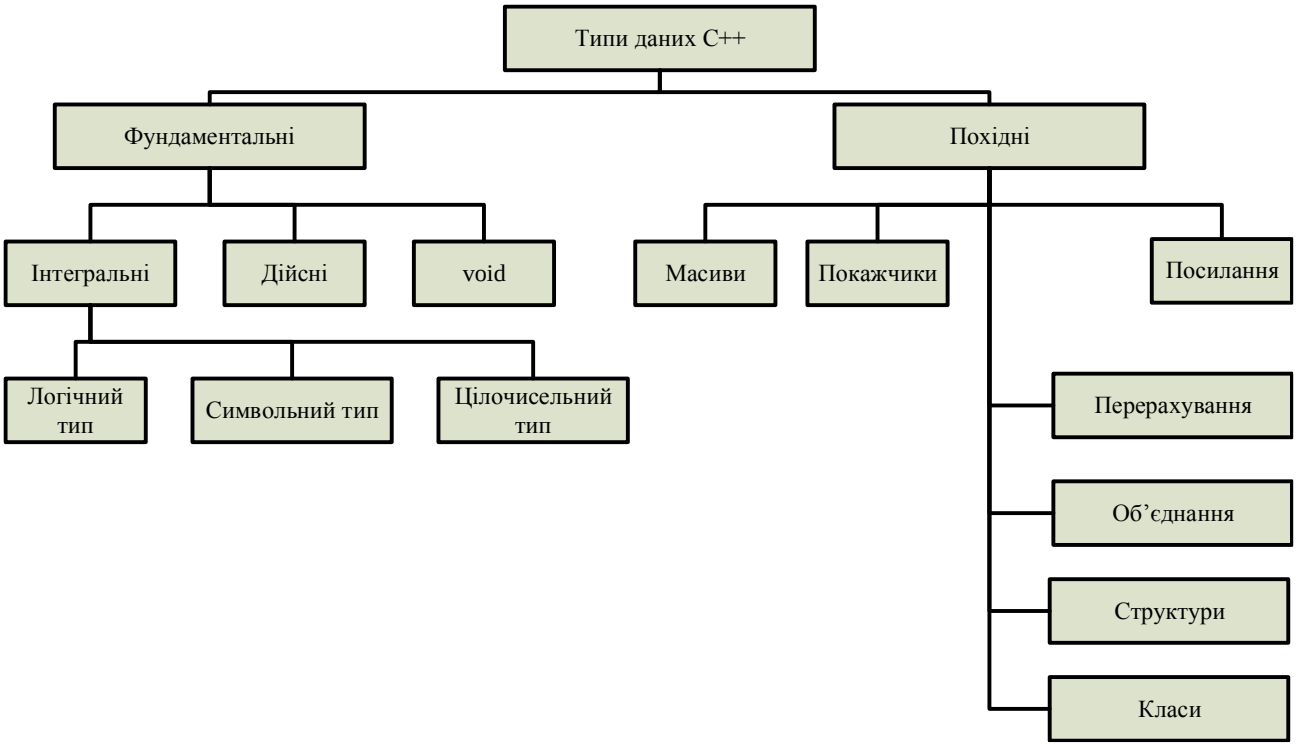

Рис. 1. Класифікація типів даних С++

У мові С++ визначено такі базові типи даних:

bool: логічний тип. Може приймати одно з двох значень true (істинно) та false (хибно).

Символьні типи

char: представляє один символ кодування ASCII. Займає у пам'яті 1 байт (8 біт). Може зберігати будь-яке значення з діапазону від -128 до 127 або від 0 до 255.

Змінна типу char як значення приймає один символ в одинарних лапках: char c = 'd'. Також можна привласнити число із зазначеного вище у списку діапазону: char c = 120. У цьому випадку значенням змінної ***c*** буде символ, який має код 120 в таблиці символів ASCII.

signed char: представляє один символ. Займає у пам'яті 1 байт (8 біт). Може зберігати будь-яке значення з діапазону від -128 до 127.

unsigned char: представляє один символ. Займає у пам'яті 1 байт (8 біт). Може зберігати будь-яке значення з діапазону від 0 до 255.

wchar_t: являє собою розширений символ. На Windows займає у пам'яті 2 байти (16 біт), на Linux – 4 байти (32 біти). Може зберігати будь-яке значення з діапазону від 0 до 65535 (при 2 байтах), або від 0 до 4294967295 (для 4 байтів).

char16_t: представляє один символ у кодуванні Unicode. Займає в пам'яті 2 байти (16 біт). Може зберігати будь-яке значення з діапазону від 0 до 65535.

char32_t: представляє один символ у кодуванні Unicode. Займає в пам'яті 4 байти (32 біти). Може зберігати будь-яке значення з діапазону від 0 до 4294967295.

У стандарті C++11 було додано типи char16_t і char32_t, які орієнтовані на використання Unicode.

Цілочисельні типи

short (short int, signed short int, signed short): представляє ціле число в діапазоні від -32768 до 32767. Займає в пам'яті 2 байти (16 біт).

unsigned short (unsigned short int): представляє ціле число в діапазоні від 0 до 65535. Займає в пам'яті 2 байти (16 біт).

int (signed int, signed): представляє ціле число. Залежно від архітектури процесора може займати 2 байти (16 біт) або 4 байти (32 біти). Діапазон граничних значень відповідно також може варіюватися від -32768 до 32767 (при 2 байтах) або від -2147483648 до 2147483647 (при 4 байтах). Але в будь-якому випадку розмір повинен бути більшим або дорівнювати розміру типу short і менше або дорівнювати розміру типу long.

unsigned int (unsigned): представляє ціле позитивне число. Залежно від архітектури процесора може займати 2 байти (16 біт) або 4 байти (32 біти), і через це діапазон граничних значень може змінюватися: від 0 до 65535 (для 2 байтів), або від 0 до 4294967295 (для 4 байтів).

long (long int, signed long int і signed long): представляє ціле число в діапазоні від -2147483648 до 2147483647. Займає в пам'яті 4 байти (32 біти).

unsigned long (unsigned long int): представляє ціле число в діапазоні від 0 до 4294967295. Займає в пам'яті 4 байти (32 біти).

long long (long long int, signed long long int і signed long long): представляє ціле число в діапазоні від −9223372036854775808 до +9223372036854775807. Займає в пам'яті, як правило, 8 байтів (64 біти).

unsigned long long (unsigned long long int): представляє ціле число в діапазоні від 0 до 18446744073709551615. Займає в пам'яті, як правило, 8 байтів (64 біти).

Типи чисел з плаваючою крапкою (дійсні)

float: представляє дійсне число ординарної точності з плаваючою крапкою в діапазоні +/- 3.4E-38 до 3.4E+38. У пам'яті займає 4 байти (32 біти).

double: представляє дійсне число подвійної точності з плаваючою крапкою в діапазоні +/- 1.7E-308 до 1.7E+308. У пам'яті займає 8 байтів (64 біти).

long double: представляє дійсне число подвійної точності з плаваючою точкою не менше 8 байтів (64 біти). Залежно від розміру пам'яті може відрізнятися діапазон допустимих значень.

void: тип без значення.

Специфікатор auto

Іноді буває важко визначити тип виразу. І згідно з останніми стандартами можна надати компілятору самому виводити тип об'єкта. І

для цього використовується специфікатор auto. При цьому якщо визначається змінна зі специфікатором auto, ця змінна має бути обов'язково ініціалізована будь-яким значенням:

auto number = 5;.

На підставі значення компілятор виведе тип змінної. Неініціалізовані змінні зі специфікатором auto не допускаються:

auto number;.

Операції, які застосовуються до будь-якого базового типу даних

Операції у порядку зменшення пріоритету:

найвища категорія:

[] опис масиву;

:: дозвіл області видимості;

+ унарний плюс;

- унарний мінус;

++ інкрементування;

-- декрементування;

& взяття адреси;

sizeof розмір операнда в байтах;

приведення (перетворення) типу;

* множення;

/ ділення;

+ бінарний плюс;

- бінарний мінус;

<менше;

<= менше чи дорівнює;

> більше;

>= більше чи дорівнює;

== дорівнює;

!= не дорівнює;

тернарна операція;

= присвоїти;

*= помножити та присвоїти;

/= поділити та присвоїти;

+= додати та присвоїти;

-= відняти і присвоїти;.

Операції над даними інтегрального типу

Згадаймо, що за прийнятою класифікацією до даних інтегрального типу належать:

- булевий тип;
- символьні типи;
- цілочисельні типи.

Операції:

! логічне заперечення;
~ звернення (інвертування біта);
% виділення залишку від ділення;
<< побітовий зсув вліво;
>> побітовий зсув праворуч;
& побітове “І”;
^ побітове виключне “АБО”;
| побітове "АБО";
&& логічне “І”;
|| логічне "АБО";
%= взяти залишок та присвоїти;
&= виконати побітове “І” та присвоїти;
^= виконати побітове виключне “АБО” і присвоїти;
|= виконати побітове "АБО" і присвоїти;
<<= виконати зсув вліво і присвоїти;
>>= виконати зсув праворуч і присвоїти;
= присвоїти;
*= помножити та присвоїти;
/= розділити та присвоїти;
+= скласти та присвоїти;
-= відняти і присвоїти.

Також є дві унарні арифметичні операції, які проводяться над одним числом: ++ (інкремент) та -- (декремент). Кожна з операцій має два різновиди: префіксний та постфіксний:

Префіксний інкремент: збільшує значення змінної на одиницю та отриманий результат повертає.

Приклад:

```
int a = 8;
int b = ++a;
cout << a << "\n"; // 9
cout << b << "\n"; // 9
```

Постфіксний інкремент: повертає значення змінної, потім збільшує на одиницю.

Приклад:

```
int a = 8;
int b = a ++;
cout << a << "\n"; // 9
cout << b << "\n"; // 8
a++; //10
++a; //11
```

Префіксний декремент: зменшує значення змінної на одиницю та отриманий результат повертає.

Приклад:

```
int a = 8;
```

```
int b = --a;
cout << a << "\n"; // 7
cout << b << "\n"; // 7
```

Постфіксний декремент: повертає значення змінної, потім зменшує на одиницю.

Приклад:

```
int a = 8;
int b = a--;
cout << a << "\n"; // 7
cout << b << "\n"; // 8
a--;
--a;
```

Порозрядні операції

Порозрядні операції також проводяться тільки над відповідними розрядами цілих операндів:

- &: порозрядна кон'юнкція (операція І чи порозрядне множення). Повертає 1, якщо обидва з відповідних розрядів обох чисел дорівнюють 1;
- |: порозрядна диз'юнкція (операція АБО або порозрядне додавання). Повертає 1, якщо хоча б один із відповідних розрядів обох чисел дорівнює 1;
- ^: порозрядне виключне АБО. Повертає 1, якщо тільки один із відповідних розрядів обох чисел дорівнює 1;
- ~: розрядне заперечення, або інверсія. Інвертує всі розряди операнда. Якщо розряд дорівнює 1, він стає 0, і якщо він дорівнює 0, він отримує значення 1.

Побітові операції виконуються над окремими розрядами чи бітами чисел. Ці операції здійснюються лише над цілими числами.

У таблиці 1 наведено приклади використання порозрядних операцій.

Таблиця 1

Приклади використання порозрядних операцій

a	b	a&b	a|b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

```
int a=8;
int b=11;
int c = a ^ b; // 1000 ^ 1011 = 0011
```

Операції зсуву

Кожне ціле число у пам'яті подано у вигляді певної кількості розрядів. І операції зсуву дозволяють зрушити бітове уявлення числа на кілька розрядів вправо чи вліво. Операції зсуву застосовуються тільки до цілих операндів. Є дві операції:

- << Зсуває бітове уявлення числа, представленого першим операндом, вліво на певну кількість розрядів, що задається другим операндом.
- >> Зсуває бітове уявлення числа вправо на певну кількість розрядів.

Приклад:

```
int c;
// зсув вліво
c = 4 << 1; // 100 ->1000 ->8
// зсув вправо
c = 16 >> 1; // 10000->1000->8
```

Умовні конструкції

Умовний оператор

```
if(вираз-умова)
    інструкція_1
[else
    інструкція_2]
```

Конструкція switch

Іншу форму організації розгалуження програм представляє конструкція switch ... case. Вона має таку форму:

```
switch (вираз){
   case значення_1: оператори_1;
   case значення_2: оператори_2;

   default: оператори;
}
```

Після ключового слова switch у дужках йде порівнюваний вираз. Значення цього виразу послідовно порівнюється зі значеннями після оператора case. І якщо збіг буде знайдено, то виконуватиметься певний блок case.

Наприкінці конструкції switch може стояти блок default. Він необов'язковий і виконується в тому випадку, якщо значення після switch не відповідає жодному оператору case.

Тернарний оператор

Тернарний оператор ?: дозволяє скоротити визначення найпростіших умовних конструкцій if і має таку форму:

[перший операнд – умова]? [другий операнд] : [третій операнд];.

Оператор використовує відразу три операнди. Залежно від умови тернарний оператор повертає другий чи третій операнд: якщо умова дорівнює true (тобто істинно), то повертається другий операнд; якщо умова дорівнює false (тобто хибно), то третій.

Приклад:

```
.#include <iostream>
 int main()
{
   int x = 5;
   int y = 3;
   char sign;
   cout << "Введіть знак операції: ";
cin >> sign;
   int result = (sign=='+') ?( x + y) : (x – y);
   cout << "Результат: " << result << "\n";
   return 0;.
```

Цикли

Для виконання деяких дій багато разів залежно від певної умови використовуються цикли. У мові C++ є такі види циклів:

- for;
- while;
- do...while;
- foreach.

Цикл while

Цикл while виконує деякий код, поки його умова є істинною, тобто повертає true:

```
while (умова)
{
    // виконувані дії
}
```

Приклад:

```
int i = 1;
        while(i < 10)
        {
          cout<<"i="<< i<<endl;
           i++;
        }
```

Цикл for

Цикл for має таке формальне визначення:

```
 for (ініціалізація; умова; модифікація_лічильників)
{
    // тіло циклу
}
```

Приклад:

```
for(int i =1; i < 10; i++)
  {
        cout<<"i="<< i<<endl;
  }
```

Цикл do

У циклі do спочатку виконується код циклу, потім відбувається перевірка умови в інструкції while. І поки ця умова істинна, тобто не дорівнює 0, цикл повторюється. Формальне визначення циклу:

```
do
{
    інструкції
}
while (умова);
```

Приклад:

```
i = 0;
  do
  {
    cout<<"i="<< i<<endl;
      i++;
  }
  while(i<10);
```

Оператори continue та break

Іноді виникає необхідність вийти із циклу до його завершення. У цьому випадку можна скористатись оператором break.

На відміну від оператора break, оператор continue здійснює перехід до наступної ітерації.

Приклад:

```
 i = 1;
  for ( ; ; )
  {
cout<<"i="<< i<<endl;
     i++;
     if (i > 9) break;
  }
```

Приклад:

```
for (int i = 0; i < 9; i++)
{
   if (i == 5)
      continue;
   cout<<"i="<< i<<endl;
}
```

Математичні функції

Для складніших математичних дій (степінь, sin, cos, корінь, …) варто підключити бібліотеку «math.h». Тобто потрібно на початку програми написати такий рядок: «#include <math.h>».

Розглянемо деякі з функцій бібліотеки:

- abs(x) – функція повертає абсолютне значення «x». Тобто це є модуль. В результаті отримуємо число з позитивним знаком: abs(3) = 3, abs(-5) = 5.
- sin(x) – приймає радіальну міру кута та знаходить його синус.
- cos(x) – приймає радіальну міру кута та знаходить його косинус.
- tan(x) – приймає радіальну міру кута та знаходить його тангенс.
- pow(x, y) – зводить число «x» в степінь «y», pow(2, 3) = 23= 8.
- sqrt(x) – повертає корінь квадратний «x», sqrt(25) = $\sqrt{25}$ = 5.
- cbrt(x) – повертає корінь кубічний з числа «x».
- «ceil(x)» – округляє до найближчого цілого числа в більшу сторону, ceil(4.2) = 5, ceil(4.6) = 5.
- floor(x) – округляє до найближчого цілого числа в меншу сторону, floor(4.2) = 4, floor(4.6) = 4.
- round(x) – округляє до цілого числа за правилами математичного округлення, round(4.2) = 4, round(4.6) = 5.
- exp(x) – обчислює експоненту числа «x».
- log(x) – обчислює логарифм натуральний (логарифм з основою експонента) від числа «x».
- log10(x) – обчислює логарифм десятковий (з основою «10») від числа «x».
- fmax(x, y) – повертає більше зі значень «x» та «y».
- fmin(x, y) – повертає менше зі значень «x» та «y».

Лекція 2. ФУНКЦІЇ. ПОКАЖЧИКИ. ОДНОВИМІРНІ МАСИВИ. СТВОРЕННЯ ГРАФІЧНОГО ІНТЕРФЕЙСУ В QT

Директива #include

#include "*ім'я файлу*"

#include <*ім'я файлу*>

Препроцесор копіює вміст файлу, що підключається в поточний файл відразу після рядка з #include.

#include <filename> дає вказівку препроцесору шукати файл у системних шляхах (у місцях зберігання системних бібліотек мови C++).

#include "filename" повідомляє препроцесору шукати файл у поточній директорії проєкту. Якщо його там не виявиться, препроцесор почне перевіряти системні шляхи.

Генерація випадкових чисел

Функція rand() є вбудованою функцією в C++ STL, яка визначена у файлі заголовка <cstdlib> (<stdlib.h>). rand() використовується для генерації ряду випадкових чисел. Випадкове число генерується за допомогою алгоритму, який видає ряд непов'язаних чисел під час кожного виклику цієї функції. Функція rand() використовується в C++ для генерування випадкових чисел у діапазоні [0, RAND_MAX).

Якщо випадкові числа генеруються за допомогою rand() без попереднього виклику srand(), програма створюватиме ту саму послідовність чисел під час кожного запуску.

RAND_MAX: це константа, значення за замовчуванням якої може відрізнятися в різних реалізаціях, але воно має бути принаймні 32767.

Приклад:

```
for (int i = 0; i < 5; i++)
        cout << rand() << " ";
```

Функція srand()

Функція srand() є вбудованою функцією в C++ STL, яка визначена у файлі <cstdlib> (<stdlib.h>). srand() використовується для ініціалізації генераторів випадкових чисел. Функція srand() встановлює початкову точку для створення серії псевдовипадкових цілих чисел.

Аргументом srand() є початкове значення для нової послідовності псевдовипадкових чисел, які повертаються послідовними викликами rand().

Генератор псевдовипадкових чисел слід задавати лише один раз перед будь-якими викликами rand() і на початку програми. Його не слід повторно заповнювати або заповнювати щоразу, коли потрібно створити нову партію псевдовипадкових чисел. Демонстрацію роботи функції srand() наведено на рис. 2.

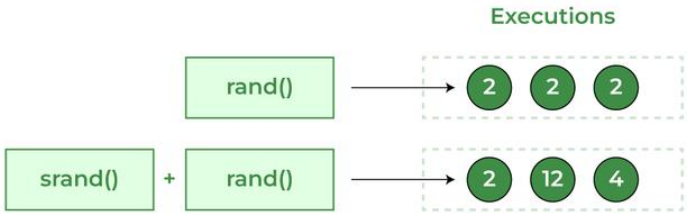

Рис. 2. Демонстрація роботи функції srand()

Стандартною практикою є використання результату виклику srand(time(0)).

time() – повертає час у секундах, починаючи з 1.01.1970 року. Для використання time() необхідно підключити бібліотеку time.h.

Приклад:

```
srand(time(0));
    for (int i = 0; i < 4; i++)
        cout << rand() << " ";
```

Є можливість масштабувати діапазон генерації випадкових чисел.

Приклад:

```
cout << rand()%3 << " "; //[0..3)
cout << rand()%3+1 << " "; //[1..4)
cout << rand()%10- rand()%10 << " ";
```

Функції

Функція – це іменована послідовність операторів. Етапи роботи з функціями: оголошення, визначення і виклик функції.

Перед викликом функції її необхідно оголосити. Оголошення функції ще називають прототипом. Оголошення має такий вигляд:

тип ім'я_функції (параметри);

Приклад:

```
int sum(int a, int b);

void main(){

int a1=3;
int b1=5;
int s1=sum(a1, b1);
// виклик функції (передача аргументу у функцію за значенням)
cout<< sum(8, 11) <<endl;
cout<<s1<<endl;
}

int sum(int a, int b) // визначення функції a=a1 , b=b1
{
```

```
int s=a+b;
return s;
}
```

Функції можуть не повертати значення. Щоб повідомити компілятору, що функція не повертає значення, потрібно використовувати тип повернення void.

Приклад:

```
void LinePrint();

void main(){
LinePrint();
….
}

void LinePrint()
{
cout << "Print from the function" << endl;
// функція LinePrint не повертає жодного значення, тому оператор
// return тут не потрібний.
}
```

Покажчики (вказівники)

Кожен об'єкт має адресу. Завантажуючись, кожна програма займає кілька цих адрес. Це означає, що кожна змінна та кожна функція нашої програми починається з будь-якої конкретної адреси. Адреси – це звичайні числа (0, 1, 2, 3, 4, 5) у шістнадцятковій системі обчислення.

Коли оголошується змінна певного типу, всі звернення в програмі до змінної компілятором замінюються на адресу області пам’яті, у якій зберігається значення змінної.

Операція отримання адреси: & (дозволяє отримати адреси змінних).

Приклад:

```
int main()
{// 1
int var1=10, var2=11, var3=12, var4=22;
cout<<&var1<<endl;
cout<<&var2<<endl;
cout<<&var3<<endl;
cout<<&var4<<endl;
}
```

Реальні адреси, зайняті змінними в програмі, залежать від багатьох факторів (характеристики комп'ютера, розмір оперативної пам’яті, наявність іншої програми в пам'яті тощо). Операція & дозволяє побачити

адреси в шістнадцятковому поданні – це звичайна форма запису адрес у пам'яті.

Програміст може визначити власні змінні для зберігання адрес областей пам'яті. Такі змінні називаються **покажчиками (вказівниками)**.

Таким чином, покажчики призначені для зберігання адрес області пам'яті.

У С++ розрізняють три види покажчиків: на об'єкт, на функцію і на void, що відрізняються властивостями та набором допустимих операцій. Покажчик не є самостійним типом, він завжди пов'язаний з будь-яким іншим типом.

Покажчик на функцію містить адресу, за якою розташовується код функції, тобто адреса, за якою передається управління при виклику функції. Покажчики на функції використовуються для непрямого виклику функції (не через її ім'я, а через звернення до змінної адреси, що зберігає її), а також для передачі імені функції в іншу функцію як параметр.

Вказівник (покажчик) на об'єкт містить адресу пам'яті, в якій зберігаються дані певного типу (основного або складного).

Формат оголошення: тип *ім'я;.

Приклад: int *a, b, *c;.

Покажчику на void можна присвоїти значення вказівника будь-якого типу, а також порівняти його з будь-якими покажчиками, але перед виконанням будь-яких дій з областю пам'яті, на яку він посилається, потрібно перетворити його до конкретного типу явно.

Вказівник (покажчик) на void застосовується в тих випадках, коли конкретний тип об'єкта, адреса якого вимагає зберігання, не визначений (наприклад, якщо в одній і тій самій змінній в різні моменти часу потрібно зберігати різні об'єкти).

Покажчик на об'єкт (змінну)

Комп'ютеру потрібно знати, якого саме типу змінна, на яку вказує покажчик. Покажчику обов'язково має бути надано деяке значення (покажчик обов'язково має бути проініціалізований).

Отримання значення змінної за відомою адресою:

Приклад:

```
….
int *p;
p=&var1; //присвоюємо адресу змінної var1 покажчику
// int *p=&var1; – так теж можна

cout<<*p<<endl; //показує вміст змінної через покажчик, буде 10
p=&var2;
cout<<*p<<endl; //11
*p = 37; //змінна var2 отримала значення 37
cout<<var2<<endl; //37
```

…

* – пишеться перед ім'ям покажчика, називається операцією розіменування.

*p – цей запис означає «взяти значення змінної, на яку вказує покажчик».

Покажчик на void

Доступ до значення змінної, що зберігається за адресою з використанням операції розіменування, називається непрямим доступом або розіменуванням покажчика.

Приклад:

….

```
int v;
int * pv;
pv = &v;
v=3; ⇔ *pv=3
*pv=6; ⇔ v=6
```

…

Неприпустимою є операція:

```
float f = 90.5;
int * pr1 = &f; // типи не сумісні
void * pr; //покажчик, що може вказувати на будь-який тип даних, що // необхідно, наприклад, при передачі покажчиків у функцію, яка працює // незалежно від типу даних
//….
int intvar;
float floatvar;
int *pint;
float *pfloat;
void *pvoid;

pint=&intvar;
pfloat=&floatvar;

pvoid=&intvar;
pvoid=&floatvar;
cout<<pvoid<<endl;
//….
```

Передача аргументів у функцію може бути здійснена за покажчиком:

```
void kilograms (double*) // Прототип функції
int main ()
double v = 15.7;
cout<< ”v = “<<v<<”g”<<endl;
kilograms (&v);
```

```
cout<<”вага=”<<v<<”kg”<<endl;
// 0.0157
return 0;
}
void kilograms (double * pr){
// double * pr=&v;
*pr *=0.001; // * pr те саме, що v
//*pr=*pr*0.001;}
void kilograms (double);
int main ()
double v = 15.7;
cout<< ”v = “<<v<<”g”<<endl;
kilograms (v);
cout<<”вага =”<<v<<”kg”<<endl; //15.7
return 0;
}
void kilograms (double  pr){
// double  pr=v;
pr *=0.001; //
//pr=pr*0.001;}
```

У функцію передається не копія значення змінної, їй передається посилання на цю змінну (фактична адреса змінної).

& використовується для позначення посилання на змінну та операції взяття адреси змінної.

Покажчики та модифікатор const

1) const int * cp; // покажчик на константу

Не можна змінювати значення змінної, на яку покажчик вказує, але можна змінити значення самого покажчика.

2) int * const tp; // константний покажчик (покажчик-константа)

Не можна змінити значення самого покажчика tp, але можна змінити значення того, на що він вказує. Константний покажчик завжди пов'язаний з конкретною фіксованою адресою.

```
int i; // ціла змінна
const int ci = 1; // ціла константа
int * pi; // покажчик на цілу змінну
const int * pci; // покажчик на цілу константу
int * const cp = &i; // покажчик-константа на цілу змінну
const int * const cpc =&ci; // покажчик константа на цілу константу
```

Покажчики найчастіше використовуються у роботі з динамічною пам'яттю, яка називається «купою».

«Купа» – вільна пам'ять, в якій можна під час виконання програми виділяти місце відповідно до потреб. Доступ до виділених ділянок

динамічної пам'яті, що називається динамічними змінними, здійснюється тільки через покажчики. Час життя динамічних змінних – від точки створення до кінця програми або явного звільнення пам'яті. Є різні способи роботи з динамічною пам'яттю. Один із них – це використання операцій NEW, DELETE.

Стек – це область оперативної пам'яті. Він працює в порядку LIFO (Last In, First Out), тобто останній доданий у стек шматок пам'яті буде першим у черзі на виведення зі стека. Щоразу, коли функція оголошує нову змінну, вона додається в стек, а коли ця змінна зникає з області видимості (наприклад, коли функція закінчується), вона автоматично видаляється зі стека. Коли стекова змінна звільняється, ця область пам'яті стає доступною для інших стекових змінних.

«Купа» – це сховище пам'яті, також розташоване в оперативній пам’яті, яке допускає динамічне виділення пам'яті і не працює за принципом стека: це склад для змінних. Коли виділяємо в «купі» ділянку пам'яті для зберігання змінної, до неї можна звернутися не тільки в потоці, але й у всьому додатку. Саме так визначаються глобальні змінні. Після завершення програми всі виділені ділянки пам'яті звільняються. Розмір «купи» задається під час запуску програми.

Одновимірні масиви

Масив – це скінченна іменована послідовність однотипних величин, які розташовані в пам’яті один за одним.

1. Статичні масиви:

Розмір масиву разом із типом його елементів визначають об’єм пам’яті, необхідний для розміщення масиву, що виконується на етапі компіляції. Тому розмірність може бути задана тільки позитивною цілочисельною константою.

```
float a[10];
float b[5]={30, 20, 10}; // b[0]=30, b[1]=20, b[2]=10, b[3]=0.0, b[4]=0.0
const int n=5;
int m[n]={3, 4, 5, 8, 1};
int v1[]={11, 12, 13, 14};
```

Покажчик на масив:

```
int  *p1=v1;      <= >   int  *p1=&v1[0];
```

! Ім’я масиву є його адресою. !

Цикл foreach

```
for (int &element : v1)
cout<<element<<endl;
```

З масивом можна працювати через покажчик.

Приклад:

```
int main () {
int ar [5] = {31, 54, 77, 52, 93};
```

```
for (int i = 0; i < 5; i++) {
cout << * (ar + i) << endl; // cout << ar[i] << endl;
return 0;
}
}
```

*(ar + i) – аналогічно до виразу ar[i]

*(ar + 3) – четвертий елемент масиву, ar + 3 – адреса 4-го елемента масиву.

Приклад:

```
int main () {
int a[100];
int *pa=a;
for (int i = 0; i < 100; i++) {
*pa++=i; // *pa=i; pa++;
}}
```

Передача масиву у функцію

При передачі масиву у функцію прийнято використовувати покажчики:

```
//передача масиву за покажчиком (вказівником)
    #include <iostream>
    using namespace std;
    const int MAX = 5; // кількість елементів у масиві
    void kilograms (double*); // прототип функції
    int main() {
    double varray [MAX] = {10.0, 43.1, 95.9, 58.7, 87.3};
    kilograms( varray) ; // переводимо всі елементи масиву в кілограми
    for (int j = 0; j < MAX; j++) {
    cout << " varray [ " << j << " ] = " << varray [j] << "kg" << endl;
    return 0;
    }
    // визначення функції
    void kilograms (double * ptr) { // double * ptr= varray
    for(int  j = 0;  j < MAX;  j++) {
    *ptr++ *= 0.001; // * ptr = * ptr * 0.001 ; ptr++;
    }
    }
```

Записи double* та double[ ] є еквівалентними, але синтаксис покажчиків використовується частіше.

Оскільки ім'я масиву є його адресою, то немає потреби використовувати операцію взяття адреси & при виклику функції.

2. Динамічні масиви створюються за допомогою операції new, при цьому необхідно вказати тип і розмірність, що може являти собою змінну, а не константу:

Приклад:

int n=100;

float *p=new float [n]; //створюється змінна-покажчик на float.

У динамічній пам’яті виділяється безперервна область, достатня для розміщення 100 елементів типу float, а адреса її початку записується в покажчик p. Динамічні масиви не можна ініціалізувати під час створення.

Виділення ділянки динамічної пам’яті і присвоювання її адреси покажчику

int *n=new int;

Операція new виконує виділення ділянки динамічної пам'яті, достатньої для розміщення величини типу int, і записує адресу початку цієї ділянки в змінну n. Пам’ять під саму змінну виділяється на етапі компіляції.

cout<<”*n=”<<*n<<endl; //0

*n=12;

cout<<”*n=”<<*n<<endl; //12

int *m=new int (10);

Операція new виконує виділення ділянки динамічної пам'яті, достатньої для розміщення величини типу int, і записує адресу початку цієї ділянки в змінну m. Також виконується ініціалізація виділеної динамічної пам’яті значенням 10.

cout<<”m=”<<m<<endl; //12d07040

cout<<”*m=”<<*m<<endl; //10

int *q=new int [10];

Операція new виконує виділення пам’яті під 10 величин типу int (масив із 10 елементів) і записує адресу початку цієї ділянки в змінну q, що може трактуватися як ім’я масиву. Через ім’я можна звернутися до будь-якого елементу масиву.

Арифметичні операції з покажчиками

Арифметичні операції з покажчиками (додавання з константою, віднімання, інкремент і декремент) автоматично враховують розмір типу величин, адресованих покажчиками.

Ці операції застосовні до покажчиків тільки одного типу і мають сенс переважно при роботі зі структурами даних, послідовно розміщених у пам'яті, наприклад, з масивами.

Інкремент переміщує покажчик до наступного елементу масиву, *декремент* – до попереднього. Фактично значення покажчика змінюється на величину sizeof (тип). Якщо покажчик на певний тип збільшується або зменшується на константу, його значення змінюється на величину цієї константи, помножену на розмір об'єкта цього типу, наприклад:

short * p = new short [5];

p++; // значення p збільшується на 2

```
long * q = new long [5];
q++; // значення q збільшується на 4
int i = 70;
int * k = &i;
int *kk = k + 3;
cout << “kk” << kk << endl;
cout << kk – k;
```

Різниця двох покажчиків – це різниця їх значень, поділена на розмір типу в байтах.

Додавання двох покажчиків не допускається.

При записі виразів із покажчиками слід звертати увагу на пріоритети операцій. Як приклад, розглянемо послідовність дій, задану в операторі:

```
int *q=new int [10];
for (int i=0; i<10; i++)
*p++ = i;
//*p = i; p++;
```

Операції *розадресації* та *інкременту* мають однаковий пріоритет і виконуються справа наліво, але, оскільки інкремент постфіксний, він виконується після виконання операції присвоювання. Таким чином, спочатку за адресою, записаною в покажчику p, буде записано значення 10, а потім покажчик буде збільшений на кількість байтів відповідно до його типу. Те саме можна записати докладніше:

```
*p = 10; p++;
```

Операція sizeof (тип) – розмір типу в байтах.

Приклад. Консольний додаток:

```
int main(){
int n;
cout<<"Input n:"<<endl;
cin>>n;
float *m=new float [n];
srand(time(0));
for (int j=0; j<n; j++)
m[j]=(rand()%21)/2-5;  //
float sum=0;
for (int i=0; i<n; i++)
if (m[i]>0) sum=sum+m[i]; // sum+=m[i]
cout<<"sum="<<sum<<endl;
}
```

Програми з графічним інтерфейсом

Фреймворк Qt

Qt є крос-платформним (не залежить від платформи/ОС) фреймворком для розроблення додатків мовою C++. За допомогою Qt

були розроблені такі відомі програми, як: KDE, Opera, Google Earth та Skype. Вперше Qt було опубліковано у травні 1995 року.

Qt5 передбачає подвійне ліцензування, а це означає, що Qt може бути використаний з некомерційною метою для створення програм із відкритим вихідним кодом, а також як ліцензія для комерційних клієнтів. За рахунок використання власного фреймворку та потужного інструментарію Qt дозволяє швидко та зручно створювати власні крос-платформні програми.

Модель подій у програмах Qt5

Механізм сигналів і слотів є розширенням мови програмування C++ Qt5, який використовується для встановлення зв'язку між об'єктами. Якщо відбувається певна подія, то при цьому може генеруватися сигнал. Цей сигнал потрапляє у пов'язаний з ним слот. Слот — це звичайний метод у мові C++, який приєднується до сигналу; він викликається тоді, коли генерується зв'язаний із ним сигнал.

Всі графічні програми керуються подіями: все, що відбувається в додатку, є результатом обробки тих чи інших подій. Вони є важливою частиною будь-якої графічної програми. Здебільшого події генеруються користувачем програми, але вони також можуть бути згенеровані іншими засобами, наприклад, підключенням до інтернету, віконним менеджером або таймером.

Віджет QLabel

Віджет QLabel використовується для відображення тексту або зображення. При цьому варто зазначити, що у нього не передбачено взаємодії з користувачем.

Віджет QTableWidget

QTableWidget — це унікальний віджет, який використовується у програмах для роботи з електронними таблицями (його ще називають «віджетом сітки»). Це один із найскладніших віджетів.

Віджет QLineEdit

Віджет QLineEdit є редактором однорядкового тексту. Редактор рядка дозволяє користувачеві вводити один рядок звичайного тексту і використовувати такі функції редагування, як: скасування, повтор, вирізання, вставка, а також перетягування за допомогою механізму drag-and-drop.

На рис. 3 наведено графічний інтерфейс створюваного додатка на Qt для роботи з одновимірними масивами.

Приклад. Додаток з графічним інтерфейсом:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QTableWidget>
#include <QMessageBox>
#include <stdlib.h>
#include <time.h>
```

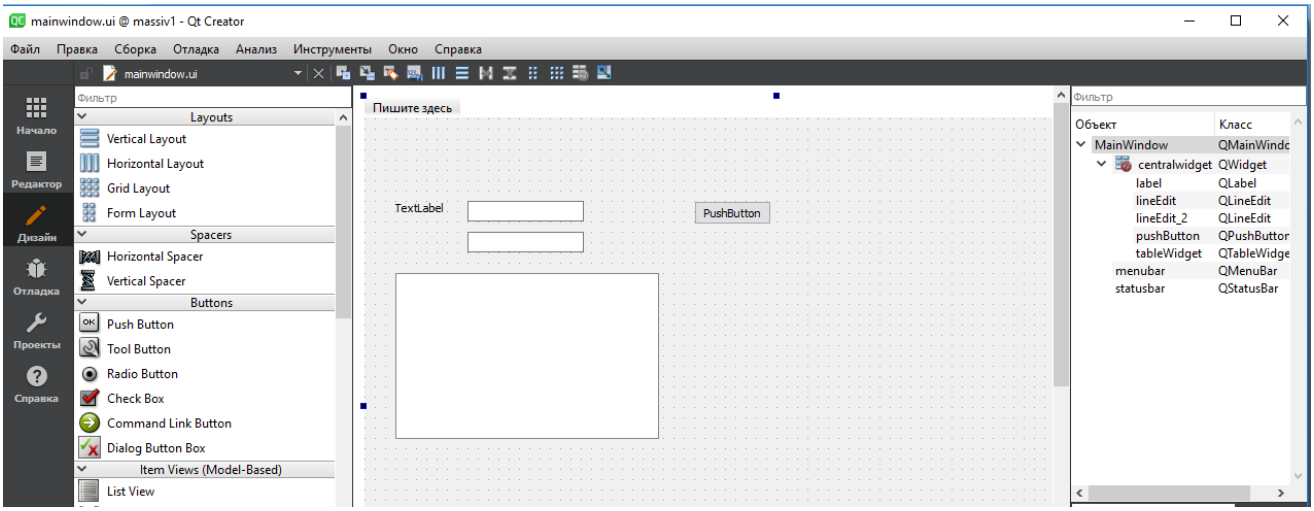

Рис. 3. Графічний інтерфейс створюваного додатка на Qt

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    srand(time(0));
    ui->label->setText("кількість елементів");
    ui->lineEdit->setText("");
}

MainWindow::~MainWindow()
{
    delete ui;
}

int* m;
void MainWindow::on_pushButton_clicked()
{
  // ui->label->clear();
 // ui->tableWidget->show();
 // QString v1, r;
  int n=(ui->lineEdit->text()).toInt();
  m=new int[n];

  ui->tableWidget->setRowCount(n);
  ui->tableWidget->setColumnCount(1);
  int sum=0;
        for(int i = 0; i<n; i++)
```

```
        {
            QTableWidgetItem *item = new QTableWidgetItem();

            m[i] = rand()%20;
            item->setText(QString::number(m[i]));
            ui->tableWidget->setItem(i,0,item);
            sum+=m[i];

        }

        ui->lineEdit_2->setText(QString::number(sum));
}
```

Лекція 3. ПОСИЛАННЯ

Посилання є псевдонімом, альтернативним ім'ям змінної, також синонімом імені.

Найважливіша властивість – передача аргументів у функцію.

Формат оголошення посилання:

Тип & ім'я;

де тип – це тип величини, на яку вказує посилання;

& – оператор посилання, що означає, що наступне за ним ім'я є ім'ям змінної типу посилання, наприклад:

```
int k;
int & p = k; // посилання p – альтернативне ім'я для k
```

Правила використання посилань:

1. Посилання має явно ініціалізуватися при його описі, крім випадків, коли воно є параметром функції.
2. Після ініціалізації посиланню не може бути присвоєно іншу змінну.
3. Тип посилання повинен збігатися з типом величини, на яку вона посилається.
4. Не дозволяється визначати вказівники (покажчики) на посилання, створювати масиви посилань і посилання на посилання.
5. Посилання, на відміну від покажчика, не займає додаткового простору в пам'яті і є просто іншим ім'ям величини.
6. Операція над посиланням призводить до зміни величини, на яку вона посилається.
7. Посилання застосовуються найчастіше як параметри функцій і типів значень, що повертаються з функцій [4].

Приклади:

```
void f ()
{
int n =1;
int & r=n; // r і n посилається на одне і те саме ціле
int x = r; // x=1
r=2; // n=2
n=4; // r=4
}

void interment (int & aa)
// int & aa=x
{aa++;}
int main()
{
int x =1;
interment (x);} // x=2
```

```
void interment (int  aa)
// int  aa=x =>aa=1
{aa++;}
int main()
{
int x =1;
interment (x);} // x=1
```

Посилання може використовуватися як аргумент функції, яка змінює значення об'єкта, що передається їй.

Існують 3 способи передачі аргументу на функцію:

1. Спосіб передачі аргументів, за якого функція створює копії переданих значень, називається передачею аргументів за значенням. Таким чином, функція не має доступу до змінних-аргументів, а працює зі зробленими їй копіями значень.

Такий механізм корисний, коли функції немає необхідності змінювати значення аргументів, і здійснюється захист аргументів від несанкціонованого доступу.

2. Важливою особливістю передачі аргументів за посиланням є те, що функція має прямий доступ до значень аргументів і може їх змінювати. До переваг також відноситься можливість повернення функцією програмі не одного, а безлічі значень (посилальний механізм).

3. Під час передачі аргументів за вказівником (покажчиком) функція також має прямий доступ до значень аргументів і може їх змінювати.

Приклади:

```
void Func(float n, float m, float &pr, float &r) // pr, r – псевдоніми
 // для тих змінних, що будуть передані у функцію
{
// n=k, m=t, float &pr=d, float &r=v
pr=n*m;
r=n-m;
}

void changeFunc(int &s, int &q)
{
// int &s=c, int &q=f
int temp=s;
s=q;
q=temp;
}

int Main() {
float  k=11.5;
float  t=-8.5;
```

```
float  d, v;
Func(k, t, d, v);
cout<<”d=”<<d<<endl;
cout<<”v=”<<v<<endl; //20

int c=13, f=38;
changeFunc (c, f);
cout<<”c=”<<c<<endl; // 38
cout<<”f=”<<f<<endl; //13
            }
```

Константні аргументи функції

Посилальний механізм передачі аргументів у функцію дозволяє функції змінювати значення аргументів. У цій програмі функція Func() не зможе змінити значення змінної "b":

```
void Func( int &a, const int &b );
int main() {
   int a = 7;
   int b = 11;
   Func( a, b );
   return 0;
}
void Func( int &a1, const &b1 ) {
   a1 = 107;
   b1 = 111; // помилка при спробі змінити константний аргумент
}
```

Приклади:

```
int func(int w)
{
// w=15
w*=10; //w =w*10
return w;
}

void func_(int w)
{
w*=10;
cout<<”func_:”<< w<<endl;
}

void func1_(int *x) // int *x=&w
{
*x*=10; // *x=*x*10
}
```

```
void func2_(int & x)  // int & x=w
{
x*=10;
}
int main() {
int w=15;
//передача аргументу за значенням
cout<<func(w)<<endl; //150
cout<< w<<endl;  // 15
func_(w);  //150
cout<< w<<endl; //15
//за покажчиком
func1_(&w);
cout<< w<<endl;  //150
//за посиланням
func2_(w);
cout<< w<<endl; //1500
}
```

Покажчик на покажчик

```
int i = 70; // адреси відрізняються на 4 байти наприкінці, int = 4 байти
int *pi = &i;
int **ppi = &pi;
int ***pppi = &ppi;
cout << pppi << “    ”<<ppi << endl;
cout << “i =” << ***pppi << **ppi << endl; //70
```

Багатовимірні масиви

Багатовимірні масиви задаються зазначенням кожного вимірювання в квадратних дужках, наприклад, оператор нижче створює матрицю із 6 рядків і 8 стовпців:

```
int matr [6][8]; // 6 рядків, 8 стовпців
```

0 0	0 1	0 2	0 3	0 4	0 5	0 6	0 7
1 0	1 1	1 2	1 3	1 4	1 5	1 6	1 7
2 0	2 1	2 2	2 3	2 4	2 5	2 6	2 7
3 0	3 1	3 2	3 3	3 4	3 5	3 6	3 7
4 0	4 1	4 2	4 3	4 4	4 5	4 6	4 7
5 0	5 1	5 2	5 3	5 4	5 5	5 6	5 7

У пам'яті такий масив розташовується в послідовних комірках рядках. Для доступу до елемента багатовимірного масиву вказуються всі його індекси, наприклад, matr[i][j], або більш екзотичним способом: *(matr[i] + j) або *(*(matr + i) + j). Це можливо, оскільки matr[i] є адресою початку і-го рядка масиву.

Приклад:

```
…
for (int i=0; i<6; i++) {
for (int j=0; j<8; j++)  {
matr[i][j]=i*j; // *(matr[i] + j) =i*j;        *(*(matr + i) + j) =i*j;
}}
matr[3][4]=10;
…
```

При ініціалізації багатовимірного масиву він представляється або як масив із масивів, при цьому кожен масив розміщується у своїх фігурних дужках (в цьому випадку ліву розмірність при описі можна не вказувати), або задається загальний список елементів у тому порядку, в якому елементи розташовуються в пам'яті:

int mass2 [] [2] = { {1, 1}, {0, 2}, {1, 0} };

1 1

0 2

1 0

int mass2 [3] [2] = { 1, 1, 0, 2, 1, 0 };

Для створення динамічного багатовимірного масиву необхідно вказати в операції new всі його розмірності (сама ліва розмірність може бути змінною), наприклад:

int n = 5;

int **m = (int **) new int [n][10];.

Більш універсальний і безпечний спосіб виділення пам'яті під двовимірний масив, коли обидві його розмірності задаються на етапі виконання програми.

Приклад:

```
…
int n, m;
cout < < " введіть кількість рядків і стовпців:";
cin >> n >> m;
int ** a = new int * [n]; // покажчик на покажчик на int і виділення пам'яті під // масив покажчиків на рядки масиву
for (int i = 0; i<n; i++) // виділення пам'яті під кожен рядок масиву
```

a[i] = new int [m]; // елементу масиву покажчиків на рядки //присвоюється адреса початку ділянки пам'яті (для стовпців)

…

Приклад. Консольний додаток:

```
#include <time.h>
#include <stdlib.h>
float Summa(float **, int, int);

int main() {
srand(time(0));
int n;
n=7;
float **a=new float *[n];
for (int i=0; i<n; i++)
a[i]=new float [n];
for (int i=0; i<n; i++) {
for (int j=0; j<n; j++)
a[i][j]=1 + rand() % 100-1 - rand() % 100; }
 float s;
s=Summa(a ,n, n); }

 float Summa(float **s, int nn, int mm)
{
  float sum=0;
  for (int i = 0; i < nn; i++) {
  for (int j = 0; j < mm; j++) {
          sum+=s[i][j];
          } }
  return sum;
}
```

Приклад. Реалізація програми з графічним інтерфейсом. Використання платформи Qt

На рис. 4 наведено графічний інтерфейс програми.

Код програми:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <ctime>
#include <stdlib.h>

float Summa(float **, int, int);

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow){
```

```
    ui->setupUi(this);
}
```

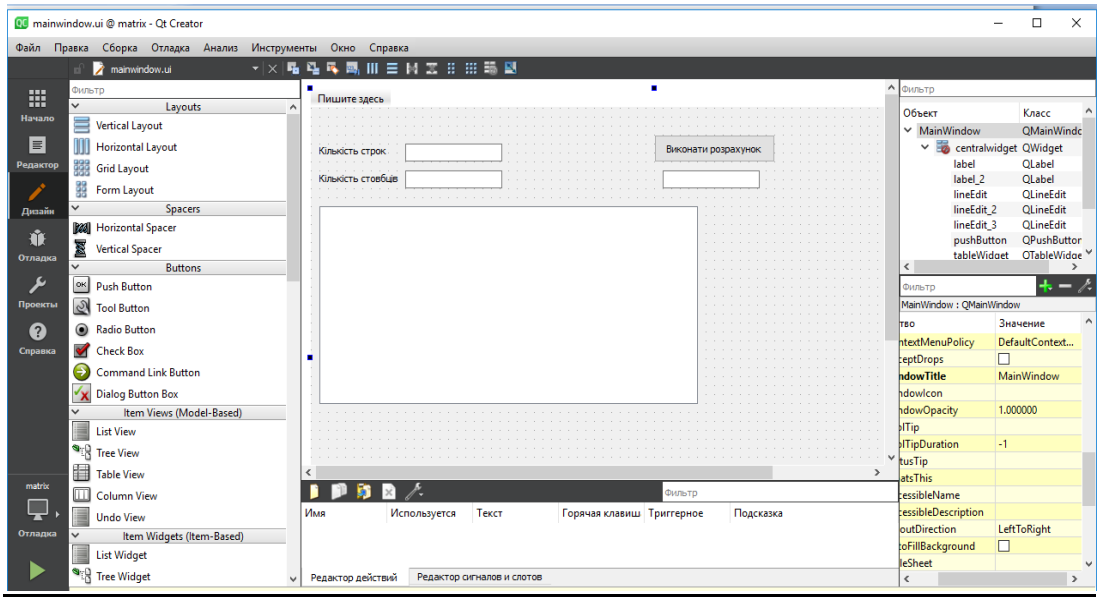

Рис. 4. Графічний інтерфейс в Qt

```
MainWindow::~MainWindow()
{
    delete ui;
}
float **a;
void MainWindow::on_pushButton_clicked()
{
   srand(time(0));
   int n=(ui->lineEdit->text()).toInt();
   int m=(ui->lineEdit_2->text()).toInt();
   ui->tableWidget->setRowCount(n);
   ui->tableWidget->setColumnCount(m);
   a=new float *[n];
   for (int i=0; i<n; i++)
   a[i]=new float [m];

   float s=0;
   for (int i=0; i<n; i++) {
   for (int j=0; j<m; j++)  {
        QTableWidgetItem *item = new QTableWidgetItem();
        a[i][j]=1 + rand() % 100-1 - rand() % 100;
            item->setText(QString::number(a[i][j]));
            ui->tableWidget->setItem(i,j,item);
         }
   }
```

```
s=Summa(a ,n, m);
        ui->lineEdit_3->setText(QString::number(s));
}

float Summa(float **s, int nn, int mm)
  {
     float sum=0;
     for (int i = 0; i < nn; i++) {
        for (int j = 0; j < mm; j++) {
        sum+=s[i][j];
        } }
     return sum;
  }
```

Приклад. Реалізація програми з графічним інтерфейсом у середовищі Visual Studio:

Код програми:

```
#include <ctime>
#include <stdlib.h>
float Summa(float **, int, int);
….
float **a;
int n;
private:  System::Void  button1_Click(System::Object^  sender,
System::EventArgs^  e) {
int::TryParse(textBox1->Text, n);
srand( time( 0 ) ); // автоматична рандомізація
a=new float *[n];
for (int i=0; i<n; i++) a[i]=new float [n];
  for (int i=0; i<n; i++) {
  for (int j=0; j<n; j++)
//*(a[i]+j)=1 + rand() % 100-1 - rand() % 100; }
//a[i][j]=1 + rand() % 100-1 - rand() % 100; }
   dataGridView1->Columns->Clear();
   dataGridView1->ColumnCount=n;
   dataGridView1->RowCount=n;
   for (int i = 0; i < n; i++)
  // {dataGridView1->Rows[i]->Name=(i+1).ToString();
     for (int j = 0; j < n; j++){
   dataGridView1->Rows[i]->Cells[j]->Value = a[i][j].ToString();
   dataGridView1->Columns[j]->Name=(j+1).ToString();
  }
  textBox4->Text=Summa(a,n,n).ToString();
  } //click
```

Лекція 4. ОПЕРАЦІЇ ЯВНОГО ПЕРЕТВОРЕННЯ ТИПІВ. ПЕРЕВАНТАЖЕННЯ ФУНКЦІЙ. ШАБЛОНИ ФУНКЦІЙ

Операції явного перетворення типів

1) Операція записується в двох форматах:

тип (вирази)

(тип) вираз

результатом операції є значення заданого типу:

```
int a=2;
float b=6.8;
cout << double (a);
cout << (int) b;
int c=(int) b;
```

2) Операція static_cast

Операція static_cast використовується для перетворення типу на етапі компіляції між:

- Цілими типами;
- Цілими і дійсними типами;

Формат операції:

static_cast <тип> (вираз).

Результат операції має вказаний тип, який може бути посиланням, покажчиком, арифметичним або перерахованим типом.

При виконанні операції внутрішнє представлення даних може бути модифіковане, хоча чисельне значення залишається незмінним.

Приклад:

```
float f = 100;
int i = static_cast <int> (f);
// Цілі і дійсні числа мають різне внутрішнє представлення
double v1=10.5;
 double v2=15.5;
int w= static_cast  <int>  (v1)+ static_cast  <int>  (v2); //25
```

3) reinterpret_cast <тип> (вираз).

Операція sizeof

sizeof вираз

sizeof (тип)

```
float x=1;
cout<<sizeof(float); //4
cout<<sizeof x; //4
cout<<sizeof (x+1.0); //8
```

Перевантаження функцій

Немає необхідності запам'ятовувати безліч імен функції. Перевантажені функції виконують різні дії, що залежать від типів даних, які

передаються функції як аргументи. Яка функція буде виконана, залежить від кількості аргументів, зазначених у виклику.

Функції з одним і тим самим ім'ям можуть мати різну кількість аргументів різних типів. Існуючі функції, наприклад, abs (fabs) – модуль числа, так само перевантажені.

Аргументи за замовчуванням

Схожий на перевантаження ефект можна досягти іншим способом – за допомогою аргументів за замовчуванням (але типи аргументів змінюватися не можуть).

Приклади:

```
void char_print();
void char_print(char);
void char_print(char, int);
void char_print(char, char);
void char_print(char, char, int);
void char_print(int, char);
void char_print_default(char='1', char='c', int=45);

int main()
{

  char_print();
  char_print('-');
  char_print('=', 15);
  char_print('+', '-');
  char_print('+', '-', 30);
  char_print(12, '*');
  char_print_default();
  char_print_default('-');
  char_print_default('=', 120); //пропустити параметр не можна,
//виводиться код символу 120
  char_print_default('=','-');
  char_print_default('-','*', 30);
  return 0;
}

void char_print()
{
for (int i = 0; i < 45; i++)
  cout<<'*';
  cout<<endl;
}
```

```
void char_print(char ch)
{
for (int i = 0; i < 45; i++)
  cout<<ch;
  cout<<endl;
}

void char_print(char ch, int n)
{
for (int i = 0; i < n; i++)
  cout<<ch;
  cout<<endl;
}

void char_print(char ch, char ch1)
{
for (int i = 0; i < 20; i++)
  cout<<ch<<ch1;
  cout<<endl;
}

void char_print(char ch, char ch1, int n)
{
for (int i = 0; i < n; i++)
  cout<<ch<<ch1;
  cout<<endl;
}

void char_print(int n, char ch)
{
for (int i = 0; i < n; i++)
  cout<<ch;
  cout<<endl;
}

void char_print_default(char ch, char ch1, int n)
{
for (int i = 0; i < n; i++)
  cout<<ch<<ch1;
  cout<<endl;
}
```

Рекурсія

Рекурсія дозволяє функції викликати саму себе на виконання. Рекурсія може бути джерелом помилок. Завжди необхідна умова виходу.

Приклади:

```
// підрахунок факторіалу числа за допомогою рекурсії
#include <iostream>
using namespace std;
long func(long); // прототип
int main() {
   int n; // Число, що вводиться користувачем
   long f; // Факторіал цього числа
   cout << “ Введіть ціле число: ”;
   cin >> n;
   f= func( n );
   cout << “ Факторіал числа "< < n < < " дорівнює " < <   f  < < endl;
   return 0;
}

// функція func()
// рекурсивно підраховує факторіал числа
long func( long n ) {
   If ( n > 1)
      return n * func( n -1 ); // виклик самої себе
   else
      return 1;
}
```

Зверніть увагу на те, що кожен екземпляр функції зберігає своє значення параметра n під час виконання рекурсивного виклику.

Шаблони функцій

У мові C++ шаблони функцій – це функції, які є зразком для створення інших подібних функцій. **Головна ідея** – створення функцій без вказівки точного типу(-ів) деяких чи всіх параметрів. Для цього визначаємо функцію, вказуючи тип шаблону, який використовується замість будь-якого типу даних. Після того, як створено функцію з типом параметра шаблону, фактично створюється трафарет функції.

При виклику шаблону функції компілятор використовує «трафарет» як зразок функції, замінюючи тип параметра шаблону на фактичний тип змінних, що передаються у функцію.

Всі шаблони функцій починаються зі слова template, після якого йдуть кутові дужки (<>), в яких перераховується список параметрів. Кожному параметру має передувати зарезервоване слово typename або class. Потім називаємо тип параметра шаблону (зазвичай T).

Шаблони функцій, власне кажучи, – це вказівки, згідно з якими створюються локальні версії шаблонної функції для певного набору параметрів і типів даних. Насправді, шаблони функцій – це потужний інструмент C++, який набагато спрощує роботу програміста.

Приклад 1:

```
#include <iostream>
using namespace std;
template <typename T> // оголошення параметра шаблону функції
T abs(T n){
// замість T підставляється конкретний тип даних під час виклику шаблонної функції
return (n<0)? -n : n;
}

int main(){
int  i = 5;
double d = -10.15;
//виклик функції
cout << “\n abs(“<<i<<”)=” << abs(i); // abs(int)
cout << “\n abs(“<<d<<”)=” << abs(d); // abs(double)
cout << endl;
return 0;
}
```

T – параметр шаблону. T – це не тип даних, це зарезервоване місце під будь-який вбудований тип даних. Тобто коли виконується виклик цієї функції, компілятор аналізує параметр шаблонованої функції та створює екземпляр для відповідного типу даних: int, char тощо. Генерація коду не відбувається доти, доки функція не буде реально викликана під час виконання програми. Коли компілятор бачить такий виклик функції, він знає, що потрібно використовувати тип int, оскільки це є тип аргументу і, переданого в шаблон значення (попередній приклад). Тому він генерує код abs для типу int, підставляючи його замість T. Це називається реалізацією шаблону функції, у якій кожен реалізований шаблон функції називається шаблонною функцією.

Приклад 2:

```
template <typename T> void my_swap (T & first, T & second)
{
T temp(first); // T temp=first;
first = second;
second = temp;}

int main () {
```

```
int a = 5;
int b = 10;
my_swap<int> (a, b);
double c = 77.89;
double d = 54.22;
my_swap <double> (c, d); // my_swap (c, d);
std::cout << c << “ ” << d << std::endl;
}
```

Аргументи шаблону мають бути узгоджені.

Приклад 3:

```
template < typename atype>
int find (atype* array, atype value, int size){
for (int j = 0; j < size; j++){
if (arra ‘y[j] == value)
        return j;
return -1;
}
}
….
int intarray[] = {1,3,5,7}; // масив int
float fl = 5,0; // значення float
int value = find(intarray, fl, 4) // так не можна
…
```

Приклад 4:

Якщо потрібно кілька типів параметрів шаблону, то вони розділяються комами:

```
template <typename T1, typename T2>
// Шаблон функції

template < typename atype, typename btype>
btype find (atype* array, atype value, btype size)
{
for (btype j = 0; j < size; j++) {
if (array[j] == value)
        return j;
}
return static_cast<btype> (-1);
}
```

STL (Standard Template Library) – стандартна бібліотека шаблонів містить багато шаблонів класів і функцій.

Приклад 5:

```
template < typename T1>
T1 sum(T1 *array, int size)
```

```
{
T1 sum_array = 0;
for (int i=0; i<size; i++)
sum_array+=array[i];
return sum_array;
}
int main(){
int n=10;
srand (time (0)); // автоматична рандомізація
unsigned *mas=new unsigned [n];
for (int i=0; i<n; i++)
mas[i]=1 + rand() % 100;
cout<<sum<unsigned>(mas, n)<<endl;
float * mas_float = new float [n];
for (int i=0; i<n; i++)
mas_float[i]=1 + rand() % 100*0.15;
cout<sum(mas_float, n)<endl;
}
```

Шаблон функції – це насправді не функція, оскільки вона призводить до створення програмного коду у пам'яті. Це просто модель, трафарет для створення багатьох функцій з однієї. Це цілком гармонує із філософією ООП. Як і шаблон, клас не є чимось конкретним, це лише образ для створення безлічі схожих об'єктів.

На рис. 5 наведено графічний інтерфейс додатка, що демонструє роботу з шаблонами функцій.

Приклад:

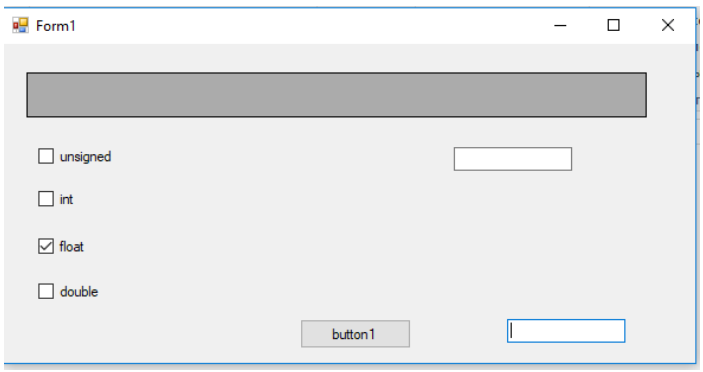

Рис. 5. Графічний інтерфейс для використання шаблонів функцій

```
template <class T1>
T1 sum(T1 *array, int size)
{
T1 sum_array=0;
for (int i=0; i<size; i++)
sum_array+=array[i];
```

```
    return sum_array;
    }

    private:    System::Void    button1_Click(System::Object^    sender,
System::EventArgs^  e) {
      int n=int::Parse(textBox1->Text);
                srand( time( 0 ) ); // автоматична рандомізація
                dataGridView1->Columns->Clear();
                dataGridView1->ColumnCount=n;
                dataGridView1->RowCount=1;
                if (checkBox1->Checked) {
                unsigned *mas=new unsigned [n];
                for (int i=0; i<n; i++)
          {
                mas[i]=1 + rand() % 100;
          dataGridView1->Rows[0]->Cells[i]->Value=mas[i].ToString();
                }
                textBox2->Text=sum<unsigned>(mas, n).ToString();
                                                        }
                 if (checkBox3->Checked) {
                float *mas_float=new float [n];
                for (int i=0; i<n; i++)
          {
                mas_float[i]=1 + rand() % 100*0.15;
     dataGridView1->Rows[0]->Cells[i]->Value=mas_float[i].ToString();
                }
                textBox2->Text=sum(mas_float, n).ToString();
                                                        }
           }
```

Лекція 5. ВКАЗІВНИКИ НА ФУНКЦІЇ

Хоча функція не є змінною, вона все одно має фізичне розташування в пам'яті, яке може бути присвоєне вказівнику. Адреса функції є вхідною точкою функції. Тому для виклику функції можна використовувати вказівник на функцію.

У деяких програмах користувач може вибрати якийсь варіант зі списку можливих дій. Наприклад, система підрахунку може мати меню з-понад 20 варіантами. Коли вибір зроблений, програма виконує відповідну функцію. Зробити це можна двома способами. Зазвичай для цієї мети використовується оператор switch. Однак у додатках, що вимагають високої продуктивності, краще використовувати інший метод. Можна використовувати масив вказівників, що містить адреси функцій. Вибір, зроблений користувачем, декодується і використовується для індексації масиву вказівників, викликаючи виконання потрібної функції. Цей спосіб набагато швидший, ніж метод switch.

Використання вказівників на функцію має такий алгоритм: після того, як вказівник описаний, він прив'язується до потрібної функції, а потім виконується її виклик за допомогою вказівника. Функції, які потрібно викликати одним вказівником, повинні мати однакову сигнатуру. Нагадаємо, що термін сигнатура функції визначає специфікацію формальних параметрів функції, тобто їх видів і порядку.

Визначення вказівника на функцію має такий формат:

тип_функції(*ім'я_вказівника) (специфікація_параметрів) ініціалізатор

Тут тип функції – це тип значення, повернутого функцією.

Ім'я вказівника – ідентифікатор, за допомогою якого викликатиметься функція. Дужки та символ розіменування * є обов'язковими елементами опису.

Специфікація формальних параметрів визначає сигнатуру функції, яка може бути пов'язана із заданим покажчиком і може бути викликана через цей покажчик.

Ініціалізатор є необов'язковим елементом визначення і дозволяє пов'язати покажчик з функцією безпосередньо при її описі.

Наприклад, визначення:

```
char(*fptr)(float*,int)=func1;
або
char(*fptr)(float*,int);
fptr=func1; // func1 – ім’я функції
```

Визначається вказівник на функцію, яка повертає значення типу **char** і має два формальні параметри: вказівник на **float** і цілочисельний тип. Тут **func1** – це назва функції, з якою зв'язується вказівник.

Значення імені функції **func1** – це адреса початку функції в пам'яті. Функція **func1** повинна повертати значення типу **char** і мати два формальних параметри типу, зазначеного в описі вказівника.

Виклик:

Формат, у якому функція викликається за допомогою вказівника на функцію, може бути одним із двох наступних еквівалентних:

ім'я_вказівника(перелік_фактичних_параметрів);

(*ім'я_вказівника)(перелік_фактичних_параметрів);

Для наведеного вище визначення вказівника **fptr** виклик функції за допомогою вказівника може мати вид:

fptr(mas,5); /* або */(* fptr)(mas,5);

Звичайно, вказівник спочатку повинен бути пов'язаний з викликаною функцією.

Приклад 1

На рис. 6 наведено графічний інтерфейс програми при роботі вказівниками на функції.

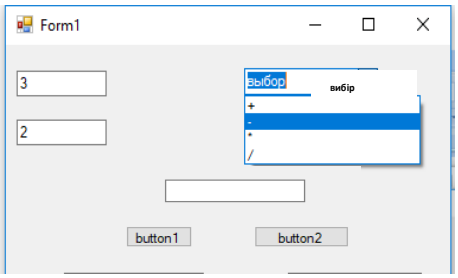

Рис. 6. Графічний інтерфейс програми при роботі вказівниками на функції

Фрагмент коду програми:

```
float add(float a, float b)  { return a+b;}
float sub(float a, float b)  { return a-b; }
float mult(float a, float b) { return a*b; }
float div(float a, float b)   { return a/b; }
float (*operation)(float, float); // визначення покажчика на функцію
float (*operMas[])(float, float)={add, sub, mult, div};

private:  System::Void  button1_Click(System::Object^  sender,
System::EventArgs^  e) {
                    float a1=float::Parse(textBox1->Text);
                    float b1=float::Parse(textBox2->Text);
                    switch (comboBox1->SelectedIndex)
                    {
                    case 0: operation=add; break;
                    case 1: operation=sub; break;
                    case 2: operation=mult; break;
                    case 3: operation=div; break;
                    }
                    textBox3->Text=operation(a1, b1).ToString();
                     }
```

```
private: System::Void button2_Click(System::Object^ sender,
System::EventArgs^  e) {
                    float a2=float::Parse(textBox1->Text);
                    float b2=float::Parse(textBox2->Text);
                    textBox3->Text=(*operMas[comboBox1->SelectedIndex])(a2,
b2).ToString(); }
```

У деяких випадках корисно об'єднувати вказівники на функції в масиви. Наприклад, цей опис:

```
void (*prtmas[3])(float);
```

Створюється масив вказівників на функції, які не повертають значень і мають єдиний формальний параметр типу float. Виклик функції за допомогою цього вказівника може мати такий вид:

```
prtmas[i](123.456);
```

Масиви вказівників зручні для програм, які використовують технологію меню.

Приклад 2

```
#include "stdafx.h"
#include "conio.h"
#include "iostream"
using namespace std;
using namespace System;
int menu();
void Enter();
void Input();
void Output();
void Exit();
void (*ptr_menu [])()={Enter, Input, Output, Exit};

int main(array<System::String ^> ^args){
  int i;
  do
  {
  i=menu();
  if (i>=1 && i<=4) ptr_menu[i-1]();
  }
  while (i>=1 && i<=4);
  _getch();
    return 0;
}
int menu()
{
  int n;
  cout<<"1 - Enter\n";
```

```
  cout<<"2 - Input\n";
  cout<<"3 - Output\n";
  cout<<"4 - Exit\n";
  cin>>n;
return n;
}
void Enter () { cout<<"function Enter is performed"<<endl; }
void Input () { cout<<"function Input is performed"<<endl; }
void Output () { cout<<"function Output is performed"<<endl; }
void Exit () { cout<<"function Exit is performed"<<endl; }
```

Лекція 6. КОДУВАННЯ. РОБОТА З РЯДКАМИ

Америка́нський інститут націона́льних станда́ртів (American National Standards Institute, ANSI) розробив версію кодування ASCII (American Standard Code for Information Interchange).

Стандарт UNICODE підтримується трьома формами (кодуванням) – 32-бітною (UTF-32), 16-бітною (UTF-16) та 8-бітною (UTF-8). Восьмибітна форма UTF-8 була розроблена для зручної сумісності з ASCII-орієнтованими системами кодування.

Найбільш простою є форма UTF-32. У ній кожен символ закодовано за допомогою 32-бітного блоку.

Стандарт UNICODE містить 96382 символи. UNICODE містить знаки пунктуації, математичні символи, технічні символи, геометричні форми та графічні позначки, фонетичні знаки: 26 букв англійського алфавіту (маленьких і великих), 9 розділових знаків (. , : ! “ ; ? ( )), пробіл, 10 цифр, 5 символів арифметичних дій (+, -, *, /, ^) і спеціальні символи (№, %, #, $, &, >, <, |, \ і т. д.). Загалом трохи більше сотні символів. Такий, порівняно невеликий, базовий набір символів можна закодувати за допомогою таблиць відповідності набору машинним кодом (фактично – двійковим числам). Символу F відповідає код 46, а символу Q – 51.

Рядки

У C++ є різні види рядків:

1) C-рядки.

Рядок є масивом символів, що закінчуються нульовим символом (нуль-символом). Нульовий символ – це символ із кодом, рівним 0, що записується як управляюча послідовність «\0». Тобто довжина рядка враховує цей символ. Рядок можна ініціалізувати рядковим літералом.

char str [10] = "рядок"; // 6 байт під символи, 1 байт під управ. послід. ("\ 0").

Розмірність можна не вказувати, якщо виконується ініціалізація:

char str [ ] = “рядок”;

char str1 [ ] = {'с', 'т', 'р', 'о', 'к', 'а'};.

Для розміщення у динамічній пам'яті треба описати покажчик на char і виділити пам'ять за допомогою операції new:

int m = 80;

char * p = new char [m];.

Операція присвоєння одного рядка іншому не визначена, оскільки такий рядок інтерпретується як одновимірний масив.

Коли операція << виводить рядок на консоль, вона виводить символи доти, поки не зустріне нульовий символ.

Для роботи з такими рядками необхідно підключати заголовочний файл <stdlib.h>.

У табл. 2 подано деякі функції для роботи з C-рядками.

Таблиця 2

Функції для роботи з рядками та символами

Функція	Пояснення
strlen(ім'я_рядка**)**	визначає довжину зазначеного рядка, без урахування нуль-символу
Копіювання рядків	
strcpy(s1,s2)	виконує побайтне копіювання символів з рядка s2 до рядка s1
strncpy(s1,s2, n)	виконує побайтне копіювання n символів з рядка s2 до рядка s1. Повертає значення s1
Конкатенація рядків	
strcat(s1,s2)	поєднує рядок s2 з рядком s1. Результат зберігається в s1
strncat(s1,s2,n)	поєднує n символів рядка s2 із рядком s1. Результат зберігається в s1
Порівняння рядків	
strcmp(s1,s2)	порівнює рядок s1 із рядком s2 і повертає результат типу int: 0 – якщо рядки еквівалентні, > 0 – якщо s1<s2, <0 – якщо s1>s2 з урахуванням регістру
Функції пошуку	
strchr(s, c)	пошук першого входження символу з у рядку s. У разі успішного пошуку повертає покажчик на місце першого входження символу с. Якщо символ не знайдено, повертається нуль
strcspn(s1, s2**)**	визначає довжину початкового сегмента рядка s1, що містить ті символи, які не входять до рядка s2
strspn**(**s1, s2**)**	повертає довжину початкового сегмента рядка s1, що містить лише ті символи, які входять до рядка s2
strprbk(s1, s2)	Повертає покажчик першого входження будь-якого символу рядка s2 у рядку s1
Функції перетворення	

Таблиця 2

Функції для роботи з рядками та символами

Функція	Пояснення
atof(s1)	перетворює рядок s1 на тип double
atoi(s1)	перетворює рядок s1 на тип int
atol(s1)	перетворює рядок s1 на тип long int

Керуючі (управляючі) послідовності

Символ \ "керує" інтерпретацією наступних за ним символів.

\n – початок наступного рядка, \r– повернення каретки, \\ – слеш, \" – лапка і т. д.

Приклад:

cout<<" \"Ну все, ми полетіли\", сказала вона.";.

Функція strlen() визначає довжину рядка, що не містить нульовий символ. При копіюванні рядків, застосовуючи цю функцію, необхідно додати нульовий символ самостійно.

Під час копіювання можна використовувати функцію strcpy(str, source).

Приклад:

```
char str1[ ] = "рядки";
const int MAX = 80; // максимальна довжина рядка
char str2[MAX];
for (int i = 0; i <strlen(str1); i++)
str2[i] = str1[i];
str2[i] = "\0";
// або
strcpy(str2, str1);
```

Приклад:

```
int main () {7 -кількість рядків у масиві, максимальна довжина якого з них 9
char str [7] [10] = {«понеділок», «вівторок»,. . . . . , "неділя"};
for (int i = 0; i < 7; i++)
cout << str[ i ] << endl;
int k=atoi(«13»);
return 0; }
```

2) Робота з об'єктами класу string

Для роботи з класом string необхідно підключити заготовочний файл <string.h>.

Створення та ініціалізація рядків класу

string(); // створює порожній об'єкт класу string

string(const char*):// створює об'єкт, ініціалізуючи його значенням С-рядка

```
string s = “test”;
string s1;
string *s2 = new string (“Hello”);
string s11 ( “ hello, “ );
string s12( “world\n” );
string s3;
char *c1= “array of string”;
s1=c1;
string *str =new string (c1);
char *str1 =s1.c_str();
```

Функція c_str() повертає покажчик на символьний масив, що містить рядок об'єкта sting у тому вигляді, в якому він перебував у вбудованому рядковому типі.

```
int n =s.size() – повертає кількість елементів у рядку.
int n =s.length() – те саме, що int n = str.size();
for (int i =0; i< n; i++)
if( str[i]==’y’) str[i]=’_’;
```

void resize(size_type count, char h) – функція змінює розмір рядка, щоб він міг містити count символів. Символ, що використовується для ініціалізації символів, що додаються наприкінці рядка (h – необов'язковий параметр).

```
s.clear() – очищає рядок.
```

Зчеплення рядків (конкатенація)

```
string s3 = s1 + s2;
s1+=s2; // s1=s1+s2
s.insert (1, "!");
cout<<s.c_str()<<endl;
```

Приклад:

```
using namespace std;
#inclucle <iostream>
#inclucle <string>
int main ( ){
string s1 ( "риба"); // str:: string
string s2 = "м'ясо";
string s3;
s3=s1;
s3="ні" + s1 + "ні";
s3+ =s2;
cout << “s3=”<<s3<<endl;
return 0;}
```

3) Клас System::String^ для роботи з рядками в середовищі Visual Studio.

Повний список методів для роботи з класом System::String^ представлений на офіційному сайті Microsoft.

```
String^ st =”рядок”;
String^ st1;
 int index=st->IndexOf(“ок”);
 if (index!=-1) {st1=st->Insert(index, “оч”);
st1=st1->Remove(index,2);
}
```

4) Клас QString для роботи з рядками у Framework Qt

Клас QString надає Unicode-рядок, який зберігає звичайний рядок як 16 бітових значень типу QChar. На відміну від рядків у багатьох інших мовах програмування, QString можна змінювати.

```
    // Ініціалізація №1:
  QString str1 = "The night train";
   out << str1 << endl;
    // Ініціалізація №2: об'єктний метод
   QString str2("A yellow rose");
   out << str2 << endl;
    // Ініціалізація №3:
   std::string s1 = "A blue sky";
   QString str3 = s1.c_str();
   out << str3 << endl;
    // Ініціалізація №4:
   std::string s2 = "A thick fog";
   QString str4 = QString::fromLatin1(s2.data(), s2.size());
   out << str4 << endl;
    // Ініціалізація №5:
   char s3[] = "A deep forest";
   QString str5(s3);
   out << str5 << endl;
QString a = "Eagle"; // Виводимо перший символ рядка
   cout << a[0] << endl;
    // Виводимо п'ятий символ рядка
   cout << a[4] << endl;
```

Є 3 методи, які дозволяють отримати інформацію про довжину рядка:

```
   size();
   count();
   length().
  for (int i = 0; i < a.size(); ++i) {
    cout << a[4] << " ";
  }
```

Конвертація рядків

Клас QString містить такі методи: toInt(), toFloat() і toLong(), які дозволяють перетворити рядок на типи int, float і long int відповідно. Метод setNum() дозволяє конвертувати різні числові дані у рядок.

```
QString s1 = "12";
QString s2 = "15";
QString s3, s4;
```

За допомогою методу toInt() конвертуються рядки в цілісний тип даних, а потім складаються числа, що вийшли:

```
cout << s1.toInt() + s2.toInt() << endl;
int n1 = 30;
int n2 = 40;
```

За допомогою методу setNum() виконується конвертація з цілого типу даних в QString, а потім вже з'єднуються рядки:

```
cout << s3.setNum(n1) + s4.setNum(n2) << endl;
```

Модифікація рядків

```
// вихідний рядок
QString str = "Lovely";
```

За допомогою методу append() додається новий рядок у кінець вихідного рядка:

```
str.append("season");
cout << str << endl;
```

За допомогою методу remove() видаляються 3 символи, починаючи з позиції 10 вихідного рядка:

```
str.remove(10, 3);
out << str << endl;
```

За допомогою методу replace() замінюються 3 символи, починаючи з позиції 7 вихідного рядка:

```
str.replace(7, 3, "girl");
cout << str << endl;
```

Очищаємо рядок:

```
str.clear();
  if (str.isEmpty()) {
    cout << "The string is empty" << endl;
  }
```

Лекція 7. СТРУКТУРИ

Структура – складний тип даних, що визначається програмістом. Після опису ставиться «;». Елементи структури називаються полями [5].

Формат опису:

```
struct [ім'я_типу] {
тип_1   елемент_1;
тип_2   елемент_2;
…
тип_n   елемент_n;
};
```

Приклад:

```
struct part
{
int  modelNumber;
int partNumber;
float cost;
};

int main(){
part   part1;
part1.modelNumber=612;
part1.partNumber=373;
part1.cost=612.0;
cout<<”Model: ”<< part1.modelNumber<<endl;
cout<<”Номер деталі: ”<< part1. partNumber <<endl;
cout<<”Вартість: ”<< part1. cost <<endl;
part   part2={143, 517, 871.6};
part   part3= part1+ part2; // error
part   part3;
part3. modelNumber = part1. modelNumber + part2. modelNumber;
part3. cost = part1. cost + part2. cost;}}
```

Приклад:

```
struct distance
{
int m;
float sm;
};

distance sum(distance, distance);
distance  d_mas[]={{4, 78}, {5, 11}, {2, 93}};
void scale(distance &, float);
void scale1(distance *, float);
```

```
void scale_mas(distance &, float,int);

struct A { int a; float x;};
struct B {A c; float y;};
B  m;
….
int main {
distance d1, d2;
          d1.m=10;
          d1.sm=24;
          d2.m=11;
          d2.sm=86;
          distance d3=sum(d1,d2);
         cout<<d3.m;
          cout<<d3.sm;

         float koef;
        cin>> koef;
         scale(d1, koef);
         cout<<d1.m<<endl;
         cout<<d1.sm<<endl;

        scale1(&d1, koef);
         cout<<d1.m<<endl;
         cout<<d1.sm<<endl;

          scale_mas(d_mas, 0.5, 3);

          distance *dp=&d1;
          int z=dp->m;
          float s=dp->sm;

          cout<<z;
          cout<<s;

         m.c.a=1;
          m.c.x=11.5;
          m.y=0.3;
          }

distance sum(distance dd1, distance dd2){
distance dd3;
dd3.m=dd1.m+dd2.m;
dd3.sm=dd1.sm+dd2.sm;
```

```
if (dd3.sm>=100) {dd3.sm-=100; dd3.m++;} // dd3.sm= dd3.sm-100;
return dd3;
}

void scale(distance & d, float k){
float d_sm=(d.m*100+d.sm)*k;
d.m=static_cast<int>(d_sm/100);
d.sm=d_sm-d.m*100;
}

void scale_mas(distance * d_mas, float k1,int n){
for (int i=0;i<n; i++) {
float d_sm=(d_mas[i].m*100+d_mas[i].sm)*k1;
d_mas[i].m=static_cast<int>(d_sm/100);
d_mas[i].sm=d_sm-d_mas[i].m*100;
}}

void scale1(distance * d, float k){
float d_sm=(d->m*100+d->sm)*k;
d->m= static_cast <int>(d_sm/100);
d->sm=d_sm-d->m*100;}
```

Лекція 8. ФАЙЛОВІ ПОТОКИ

Потік – це абстрактне поняття, що відноситься до перенесення даних від джерела до приймача.

Стандартні потоки призначені для передачі даних від клавіатури на екран (cin, cout). Файлові потоки – для обміну інформацією з файлами, а рядкові потоки – для роботи з масивами символів в оперативній пам'яті.

Класи потоків мають складну розгалужену структуру. Для підтримки потоків бібліотека C++ містить ієрархію класів, побудовану на основі двох базових класів: ios і stream. ios – базовий клас потоків.

За способом доступу файли можна розділити на послідовні, читання та запис у яких відбувається з початку байт за байтом, та файли з довільним доступом, що допускають читання та запис у вказану позицію.

Для підтримки файлового введення та виведення стандартна бібліотека C++ містить класи, зазначені в таблиці 3.

Таблиця 3

Класи файлових потоків

Клас	**Призначення**
ifstream	клас вхідних файлових потоків (для читання)
ofstream	клас вхідних файлових потоків (для запису)
fstream	клас ініціює двонаправлений файловий потік

Файлові потоки забезпечують набагато більш надійне введення / виведення, ніж старі функції бібліотеки C, що працюють із потоками типу FILE*.

Для використання файлових потоків необхідно підключити до програми заготовочний файл <fstream>.

Робота з файлом зазвичай передбачає такі операції:

- створення потоку (потокового об'єкта);
- відкриття потоку та зв'язування його з файлом;
- обмін із потоком (введення / виведення);
- закриття потоку.

Класи файлових потоків містять кілька конструкторів, що дозволяють варіювати способи створення потокових об'єктів.

Конструктори з параметрами створюють об'єкт відповідного класу, відкривають файл із зазначеним ім'ям і пов'язують файл із об'єктом.

```
ifstream (const char * name, int mode = ios :: in);
// :: – операція доступу до файлу
ofstream (const char * name, int mode = ios :: out | ios :: trunc);
fstream (const char * name, int mode = ios :: in | ios :: out);
// Режим відкриття файлу
```

Якщо значення за замовчуванням не влаштовує, можна вказати інше, вибравши одне або кілька значень (об'єднаних операцією |) із зазначених у таблиці 4.

Таблиця 4

Значення аргументу mode

Значення	**Характеристика**
ios :: in	Відкрити файл для введення
ios :: out	Відкрити файл для виведення
ios :: app	Відкрити в режимі додавання до кінця файлу
ios :: trunc	Якщо файл існує, обрізати його до нульової довжини
ios :: binary	Відкрити в двійковому режимі (за замовчуванням – текстовий режим)

Конструктори без параметрів створюють об'єкт відповідного класу, не пов'язуючи його із файлом. І тут зв'язок потоку з конкретним файлом здійснюється пізніше – викликом методу open(), який має параметри, аналогічні параметрам розглянутих вище конструкторів.

Наведемо приклади створення потокових об'єктів і зв'язування їх із конкретними файлами:

```
// файли для виведення (запису)
ofstream flog( "flog.txt");
ofstream fout1, fout2;
fout1.open(“test1”, ios :: app);
fout2.open(“test2”, ios :: binary);
 // файл для введення (читання)
ifstream finp1(“data.txt”);
//файл для введення та виведення
fstream myfile;
myfile.open(“mf.dat”);
ifstream is (“data.dat”);.
```

Методи класів файлових потоків

Клас istream – спадкоємець ios та предок для ifstream (вхідний файловий потік / читання). Методи класу ifstream наведені у таблиці 5.

Приклад 1:

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

int main() {
char buffer [80];
```

```
ifstream infile(“test.txt”);
while (  ! infile.eof()  )
{
infile.getline(buffer,  80);
cout<<buffer<<endl;
}
}
```

Метод eof() повертає нуль, якщо кінець файлу не досягнуто.

Таблиця 5

Методи класу ifstream

Метод	**Призначення**
>>	Форматований витяг даних всіх основних (і перевантажуваних) типів із потоку
get(ch)	Витягує один символ у ch
get(str)	Витягує символи str до обмежувача '\n'
get(str, MAX)	Витягує до MAX числа символів масив
get(str, DELIM)	Витягує символи до масиву str до вказаного обмежувача (зазвичай '\n'). Залишає обмежувач у потоці
get(str, MAX, DELIM)	Витягує масив str до MAX символів або символу DELIM. Залишає обмежувач у потоці
getline(str, MAX, DELIM)	Витягує масив str до MAX символів або символу DELIM. Витягує обмежувач із потоку
putback(ch)	Вставляє останній прочитаний символ назад у вхідний потік
peek(ch)	Читає один символ, залишаючи його в потоці
count=gcount()	Повертає кількість символів, прочитаних щойно методами get(), getline(), read()
read(str, MAX)	(Для бінарних файлів.) Витягує до MAX числа символів у масив str до кінця файлу (EOF)
seekg()	Встановлює відстань (у байтах) від початку файлу до файлового покажчика
seekg(pos, seek_dir)	Встановлює відстань (у байтах) від зазначеної позиції у файлі до вказівника файлу. Може приймати значення ios::beg, ios::cur, ios:: end
pos=tellg(pos)	Повертає позицію (у байтах) покажчика файлу з початку файлу

Приклад 2:

```
…
const MAX=1000;
char str[MAX];
```

```
cin.get(str, MAX, '!');
cout<<"Ви ввели: \n"<<str<<endl;
…
```

В таблиці 6 наведено методи класу ofstream.

Таблиця 6

Методи класу ofstream

Метод	**Призначення**
<<	Форматована вставка даних будь-яких стандартних (та перевантажуваних) типів
put (ch)	Вставка символу ch у потік
write (str, size)	Вставка size символів з масиву str у файл

.

Приклад 3:

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

int main() {
double  d=6.02;
string s1="bcvb";
string s2="khjk";
char c='s';
ofstream outfile("test1.txt");
outfile<<c;
 outfile<<s1;
outfile<<' ';
outfile<<s2;
outfile<<d;

cout<<"Файл записано!"<<endl;
}
}
```

Лекція 9. БІНАРНІ ФАЙЛИ

Форматоване файлове введення/виведення доцільно використовувати за невиликої кількості даних. У цьому випадку число, наприклад, 6.02, зберігається у вигляді серії символів.

Для великих послідовностей цифр ефективніше використовувати двійкове (об'єктне) введення/виведення.

Число int ->4 байти, а «12345» -> 5 байтів;

float-> 4 байти, а «6.02314513» -> 10 байтів; //серія символів

Методи для роботи з бінарними файлами

write(char * buff, int num);

buff – вказівник на буфер, що зберігає дані, які будуть надсилатися у вихідний файловий потік. Параметр num вказує число байтів у буфері, що передаються в цей потік.

read(char * buff, int num);

buff – вказівник на буфер, що приймає дані із вхідного потоку. num вказує число байтів, що зчитуються із потоку.

Приклад:

```
…
int main  {
…
int mas[100];
int mas1[100];

          for (int i=0; i<100; i++) mas[i]=i;
          ofstream write_f; //
          write_f.open("f2.dat", ios::binary);
 // ofstream write_f("f2.dat", ios::binary);
         write_f.write(reinterpret_cast<char *>(mas), 100*sizeof(int));
          write_f.close();

      ifstream read_f("f2.dat", ios::binary);
          read_f.read(reinterpret_cast<char *>(mas1), 100*sizeof(int));
          for (int i=0; i<100; i++)
         cout<<mas1[i]<<"   "<<endl;
}
```

reinterpret_cast <тип> (вираз) – операція використовується для перетворення не пов'язаних між собою типів. Дозволяє перетворювати будь-який вказівник на вказівник будь-якого іншого типу даних. Можна використовувати для перетворення char* на int*.

Розглянемо такий приклад.

Завдання

Описати структуру з ім'ям STUDENT, що містить такі поля:

- прізвище та ініціали;
- номер групи;
- успішність (масив із п'яти елементів).

Написати програму, яка виконує такі дії:

- введення з клавіатури даних у динамічний масив структур типу STUDENT;
- виведення на екран прізвищ і номерів груп для всіх студентів, які мають хоча б одну оцінку 2;
- якщо таких студентів немає, вивести відповідне повідомлення.

Приклад виконання завдання

На рис. 7 наведено графічний інтерфейс додатка, створеного засобами Framework Qt.

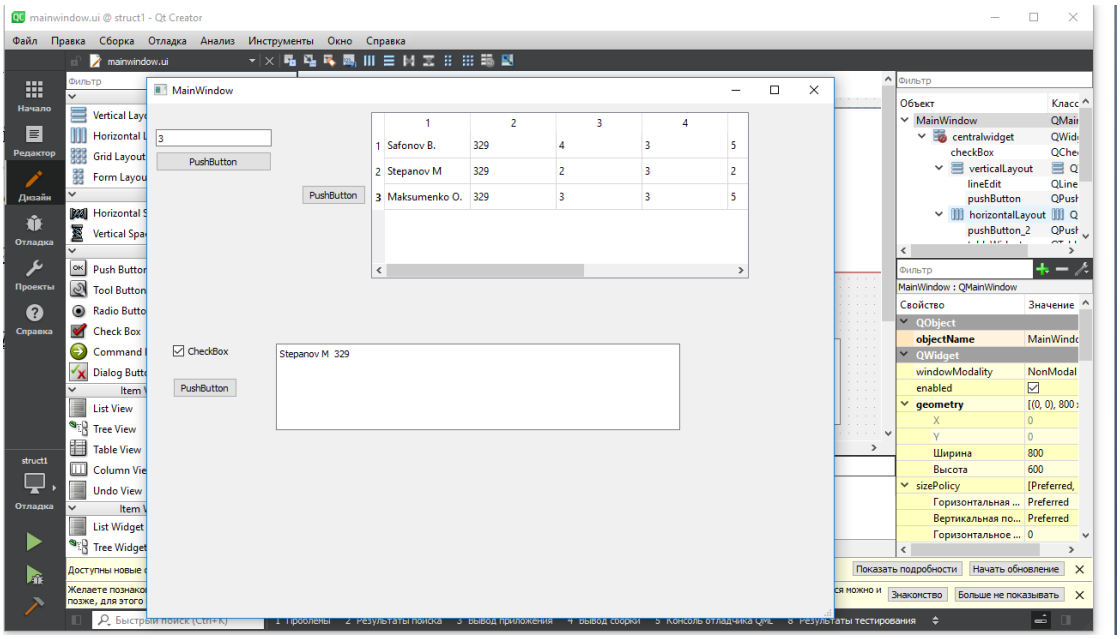

Рис. 7. Графічний інтерфейс додатка

Код програми:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QMessageBox>
using namespace std;
#include <iostream>
#include <fstream>
struct STUDENT
    {
    QString fam; //string FIO;
    int nomgroup;
```

```
        int ysp[5];
        };

bool search_2(STUDENT);

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}
MainWindow::~MainWindow()
{
    delete ui;
}

STUDENT *stud;
        int n;
STUDENT *stud_read;
void MainWindow::on_pushButton_clicked()
{
    n=(ui->lineEdit->text()).toInt();
    stud=new STUDENT [n];
    ui->tableWidget->setRowCount(n);
    ui->tableWidget->setColumnCount(7);
    stud_read=new STUDENT [n];
}

void MainWindow::on_pushButton_2_clicked()
{
ofstream outfile("D://11.dat",ios::binary|ios::out);
    for (int i=0; i<n; i++) {
    QTableWidgetItem *ItemFam;
     ItemFam  = ui->tableWidget->item(i,0);
    stud[i].fam=ItemFam->text();
    QTableWidgetItem *ItemGroup;
     ItemGroup  = ui->tableWidget->item(i,1);
    stud[i].nomgroup=(ItemGroup->text()).toInt();
          for (int j=0; j<5; j++)
          {
              QTableWidgetItem *ItemYsp;
               ItemYsp  = ui->tableWidget->item(i,2+j);
               stud[i].ysp[j]=(ItemYsp->text()).toInt();
           }
```

```
        }
         outfile.write((char*) stud, n*sizeof (STUDENT));
      }

      bool search_2(STUDENT student){
      int kol=0;
      for (int j=0; j<5; j++) if (student.ysp[j]==2) kol++;
      if (kol) return true; else return false;
      }
      void MainWindow::on_pushButton_3_clicked()
      {
         ifstream in("D://11.dat",ios::binary|ios::in);
          in.read((char*) stud_read ,n*sizeof (STUDENT));
         if(ui->checkBox->isChecked())
         {
             int h=0;
             QString str="";
             for (int i=0; i<n; i++)
             if (search_2(stud_read[i])) {
                 h++;
                str+=stud_read[i].fam+ "
"+QString::number(stud_read[i].nomgroup)+ "\n";
                                  }
             ui->textEdit->setText(str);

             if (!h) ui->textEdit->setText("Студенти з двійками не найдені!");}
          else {

          QString StringItem="";
           for (int i=0; i<n; i++) {
       StringItem+=stud_read[i].fam+ "
"+QString::number(stud_read[i].nomgroup)+ "    "+
         QString::number(stud_read[i].ysp[0])+"
"+QString::number(stud_read[i].ysp[1])+ "     "+
          QString::number(stud_read[i].ysp[2])+"
"+QString::number(stud_read[i].ysp[3])+ "     "+
          QString::number(stud_read[i].ysp[4])+ "\n";
           }

           ui->textEdit->setText(StringItem);
      }
      }
```

Робота з датами в Framework Qt

На рис. 8 наведено візуальний вигляд компонента dateEdit.

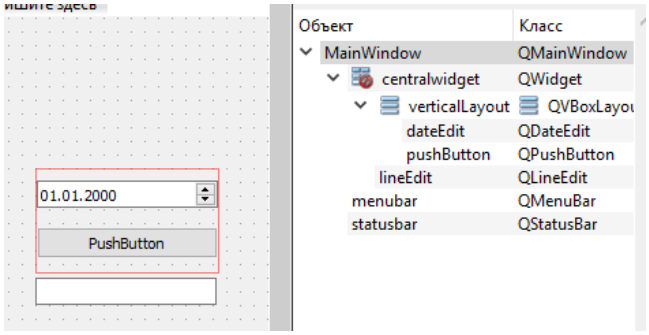

Рис. 8. Візуальний вигляд компонента dateEdit

Код програми:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QDate>
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButton_clicked()
{
    QDate Mydate= ui->dateEdit->date();
    QDate dt1(2020, 4, 10);
    QTime tm1(17, 30, 12, 54);
     //   tm1.toString("hh:mm:ss.zzz") ;
    QDate cd = QDate::currentDate(); // отримуємо поточну дату
    QTime ct = QTime::currentTime(); // отримуємо поточний час
    QDate nd = dt1.addDays(55);

   int stagD=Mydate.daysTo(cd); //in days
   int stagM=cd.month()-Mydate.month();
   int stagY=cd.year()-Mydate.year();
   ui->lineEdit->setText(QString::number(stagM+stagY*12));
}
```

Лекція 10. ОСНОВНІ ПРИНЦИПИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ (ООП)

Якщо об'єм програми збільшується, то стає неможливо утримувати всі деталі, і з'являється необхідність структурувати код, виділяти головне і відкидати несуттєве. Цей процес називається підвищенням абстракції програми.

Кроки підвищення абстракції:

1. Використання функцій. Для виклику функцій достатньо знати їх інтерфейси.
2. Опис власних типів даних, що дозволяють групувати інформацію (наприклад, структур).
3. Для робіт из власними типами даних є необхідність у створенні спеціальних функцій.
4. Об'єднання опису спеціальних типів і функцій для робіт з ними в окремих модулях.
5. Створення ієрархії класів.

Все це спрямовано на представлення програми у вигляді меншої кількості більш крупних модулів.

Класи

Клас – абстрактний тип даних, що визначається програмістом. Ідея класів – основа ООП.

Дані класу називаються полями, а функції класу – методами.

Основні властивості ООП

Інкапсуляція, наслідування, поліморфізм.

Об'єднання даних із функціями їх обробки в об`єднанні з приховуванням непотрібної для використання цих даних інформації називається інкапсуляцією. Інкапсуляція підвищує ступінь абстракції програмного коду. Дані класу і реалізація його функцій знаходяться нижче рівня абстракції, і для написання програми інформація про них не вимагається. Інкапсуляція дозволяє змінити реалізацію класу без модифікації основної частини програми [6]. Це важлива властивість, яка полягає в тому, що деталі реалізації приховані від користувачів класу інтерфейсом. Інтерфейсом класу є заголовки його методів. Інкапсуляція – це можливість маніпулювання доступом до компонентів класу за рахунок використання модифікаторів доступу.

Загальна форма опису класу

```
class <ім'я> {
[private:]
<опис прихованих елементів>
public:
<Опис доступних елементів за межами класу>
```

};

Специфікатори доступу private, public керують видимістю елементів класу. Елементи, що описані після службового слова private, «видимі» (доступні) тільки в рамках класу. Цей вид доступу прийнятий в класі за замовчуванням. Інтерфейси класу описуються після специфікатора public. Дія специфікатора поширюється до наступного специфікатора або до кінця класу. Порядок специфікаторів не має значення.

Поля класу:

- можуть мати будь-який тип, крім типу того ж класу;
- можуть бути описані з модифікатором const або static;
- ініціалізація полів при описі не допускається.

Визначення класу задає вид майбутнього об’єкту. Доступ до методів здійснюється тільки через конкретний об’єкт.

Приклад 1:

```
#include <iostream>
class smallobj {

private:
int somedata;

public:
int data2;
void setdata(int d) {
somedata=d;}

void showdata(){
cout<< «Значення поля=»<<somedata<<endl;
}};

int main(){
smallobj s1, s2;
s1.setdata(101);
s2.setdata(1700);
s1.showdata();
s2.showdata();
//s1.somedata=20; // -
s1.data2=20; //+
}
```

Конструктори

Конструктор – це метод класу, що виконується автоматично в момент створення об’єкта. Призначення конструктора – ініціалізація полів об’єкта класу.

Властивості конструкторів:

- ім’я конструктора завжди збігається з іменем класу;
- конструктор не повертає значення навіть void;
- клас може мати кілька конструкторів з різними параметрами для різних видів ініціалізації;
- конструктор, що викликається без параметрів, називається конструктором за замовчуванням;
- параметри конструктора можуть мати будь-який тип, крім того ж класу;
- якщо програміст не вказав жодного конструктора, компілятор створить його автоматично;
- конструктори не наслідуються.

Приклад 2:

```
#include <iostream>
class Counter {
private:
 int count;
public:
Counter(): count(1) { } // більш прийнятний спосіб
// Counter() {count=1;}
// ініціалізація розміщена між прототипом методу і тілом методу. Перед ініціалізацією ставиться «:», а значення, що ініціалізується, розміщується в круглих дужках після імені поля.
// int f(4); та int f=4; – є ідентичними операторами
Counter(int c): count(c) { }
void inc_count() { count++;}
int get_count()
{
return count;}

~Counter() {  cout<<”Деструктор спрацював !”<<endl; }
};

int main(){
Counter c1, c2;
c1. inc_count();
cout<< «c1=»<<c1.get_count()<<endl;
Counter c3(5);
}
```

При знищенні об’єкта викликається особливий метод – деструктор. Частіше за все слугує для вивільнення пам’яті. Деструктор не має параметрів, нічого не повертає навіть void. Ім’я деструктора збігається з іменем класу. Перед деструктором ставиться «~».

Конструктор копіювання

Приклад 3:

```
class Distance {
private:
int m;
float sm;

public:
Distance(): m(0), sm(0.0) { }
Distance(int m1, float sm1): m(m1), sm(sm1) { }

void getinfo()
{
cout<<”Enter m”;
cin>>m;
cout<<”Enter sm”;
cin>>sm;
}

void showinfo()
{
cout<<” m=”<<m<<”  sm=”<<sm;
}
};

int main(){
Distance d1(11, 6.25);
Distance d2(d1); //конструктор копіювання
Distance d3=d1; //конструктор копіювання
d1.showinfo();
d2.showinfo();
d3.showinfo();
Distance d4;
d4.getinfo();
//d4.m=10; - немає доступу
int k=8;
float d=1.1;
Distance d5(k, d);
}
```

Приклад 4:

Завдання

Описати клас з ім'ям STUDENT, що містить такі поля:

- прізвище й ініціали;

- номер групи;
- успішність (масив із п'яти елементів).

Написати програму, яка виконує такі дії:

- введення з клавіатури даних у масив об’єктів;
- виведення на екран прізвищ і номерів груп для всіх студентів, які мають хоча б одну оцінку 2;
- якщо таких студентів немає, вивести відповідне повідомлення.

Код програми

```
//файл student.h
#include <stdlib.h>
#include <string.h>
namespace STUDENT{
class student
{
private:
char fam[45];
int nomgroup;
int ysp[5];
public:
student(){strcpy(fam, "Рибенко Є."); nomgroup=329; ysp[0]=4; ysp[1]=4; ysp[2]=4; ysp[3]=4; ysp[4]=4;}
void get_student(char *, int, int, int, int, int, int);
void show();
int search_2();
void search_fio(char *);
}; }

//файл student.cpp
#include <iostream>
#include "student.h"
#include <stdlib.h>
#include <string.h>
using namespace std;
using namespace STUDENT;
void student::get_student(char * fio, int gr, int y1, int y2, int y3, int y4, int y5){
                strcpy(fam, fio);
                nomgroup=gr;
                ysp[0]=y1;
                ysp[1]=y2;
                ysp[2]=y3;
                ysp[3]=y4;
                ysp[4]=y5;
```

```
}
void student::show(){
cout<<"fam: "<<fam<<endl;
cout<<"nomgroup: "<<nomgroup<<endl;
for (int i=0; i<5; i++) cout<<"ysp"<<i+1<<":  "<<ysp[i]<<endl;
}
int student::search_2(){
int h=0;
for (int j=0; j<5; j++)
if (ysp[j]==2) h++;
return h;
}

void student::search_fio(char * fio){
if (strcmp(fio,fam)==0) {
cout<<"group: "<<nomgroup<<endl;
cout<<"ysp_1: "<<ysp[0]<<endl;
cout<<"ysp_2: "<<ysp[1]<<endl;
cout<<"ysp_3: "<<ysp[2]<<endl;
cout<<"ysp_4: "<<ysp[3]<<endl;
cout<<"ysp_5: "<<ysp[4]<<endl;
}
}

//файл class.cpp
#include <iostream>
#include <conio.h>
#include "student.h"
using namespace STUDENT;
using namespace std;
student st;
student *stud;

int main(){
int n;
cout<<" Enter kol students"<<endl;
cin>>n;
stud=new student [n];
char fio[45];
int nom_gr, ysp1, ysp2, ysp3, ysp4, ysp5;
for (int i=0; i<n; i++) {
cout<<" Enter student"<<endl;
cin>>fio>>nom_gr>>ysp1>>ysp2>>ysp3>>ysp4>>ysp5;
stud[i].get_student(fio, nom_gr, ysp1, ysp2, ysp3, ysp4, ysp5);
```

```
}

cout<<" Show students"<<endl;
for (int i=0; i<n; i++) {
stud[i].show();
}
cout<<" Show students with 2"<<endl;
int g=0;
for (int i=0; i<n; i++)
  if (stud[i].search_2()) {stud[i].show(); g++;}
if (g==0) cout<<"such students do not"<<endl;

cout<<" Enter FIO student"<<endl;
cin>>fio;
for (int i=0; i<n; i++)
stud[i].search_fio(fio);
_getch();
return 0;
}
```

Лекція 11. ПЕРЕВАНТАЖЕННЯ ОПЕРАЦІЙ

Зазвичай a = b + c; працює тільки з основними типами даних, такими, як int або float, і спроба використання звичайних операторів, коли a, b і c є об'єктами визначеного користувачем класу, призведе до протестів компілятора. Однак, використовуючи перевантаження, можна зробити цей рядок правильним навіть у тому разі, якщо a, b і c є об'єктами певного користувальницького класу.

Насправді перевантаження операцій дає можливість перевизначити мову C++.

Визначення операції, що перевантажується, оформляється у вигляді спеціальної операції-функції, в заголовку якої вказується ключове слово operator. Ця функція має такий формат:

тип_функції operator знак_операції (список формальних параметрів)
{тіло операції-функції}

Тут тип_функції – це, як і у звичайних функцій, тип значення, що повертає функція. Формальних параметрів може бути один чи два. Зазвичай формальні параметри мають тип класу, однак у деяких випадках вони можуть мати і якийсь із визначених у мові типів. У деяких випадках, що буде зазначено нижче, операція-функція може взагалі не мати формальних параметрів.

Як будь-яка інша функція, операція-функція може мати прототип. Наприклад, якщо треба визначити операцію додавання для екземплярів X та Y класу NewType, то прототип операції-функції може мати такий вигляд:

NewType operator + (NewType X, NewType Y);

Більшість операцій мови C++ можуть бути перевантажені, за винятком таких:

- «крапка» як прямий вибір компонентів структурного типу (клас.компонент);
- вибір компоненти через її покажчик (клас.*компонент);
- дозвіл області видимості (::);
- умовна операція (?:);
- операція sizeof.

Не можуть бути перевантажені препроцесорні операції #, оскільки вони і не є операціями мови, і неможливо перевантажити роздільники, наприклад, крапку з комою.

Перевантаження унарних операцій

Для перевантаження унарної операції може використовуватися функція без параметрів, звичайна або дружня функція з одним параметром.

```
class Counter
{
```

```
  private:
    unsigned int count;
  public:
    Counter() : count(0)
     { }
    Counter(int c) : count(c)
     { }
    unsigned int get_count()
     { return count; }
    void  operator++()
     {
       ++count;}
…
};

int main()
{Counter c1, c2;
  cout << "\nc1 = " << c1.get_count();
  cout << "\nc2 = " << c2.get_count();
++c1;
++c2;
}
```

Приклад 1:

```
....
    Counter  operator++()
     {
Counter temp;        // тимчасовий об’єкт Counter
temp.count = ++count;
return temp;
}
…
int main(){
Counter c3, c4;
  ++c3;
  c4 = ++c3;
}
```

Тимчасові безіменні об'єкти

У попередньому прикладі створено тимчасовий об'єкт temp типу Counter, метою якого було зберігання значення, що повертається для операції ++. Реалізація цього об'єкта вимагає трьох рядків програми:

```
Counter temp; // створюємо тимчасовий об'єкт Counter
temp.count = ++ count; // привласнюємо йому значення count
return temp; // повертаємо об'єкт
```

Існує багато шляхів повернення тимчасового об'єкта з функції або перевантаженої операції. Давайте розглянемо ще один із способів у програмі:

```
// операція ++ з використанням недекларованої змінної
#include <iostream>
using namespace std;
class Counter
{
  private:
    unsigned int count;
  public:
    Counter() : count(0)
     {}
    Counter(int c) : count(c)
     {}
    unsigned int get_count()
     {return count; }
    Counter operator++()
     {
       ++count;
       return Counter(count);
     }
// створюється об’єкт типу Counter, котрий не має імені, і він
//ініціалізується новими значеннями, але для того, щоб це було можливим, //необхідно мати конструктор з параметром у класі.
```

Приклад 2:

```
int operator++()
     {
       int k = ++ count;
       return k;
     }

Counter operator=(int i)
     {
       count = i;
       return *this;
     }
//ключове слово this трактується як покажчик на екземпляр класу
//функція класу повертає адресу об'єкта, що її викликав
};

int main()
{Counter c1, c2; // Визначаємо змінні
  cout << "\nc1 = " << c1.get_count();
```

```
  cout << "\nc2 = " << c2.get_count();
  ++c1; // збільшуємо c1
  c2 = ++c1; // c1 = 2, c2 = 2
int count1 = ++ c1;
int k=5;
c1=k;
}
```

Для виконання наведеного програмного коду клас має вміщувати у собі конструктор з одним аргументом.

Перевантаження постфіксних операцій

Відмінність між цими функціями лише тому, що у дужках стоїть int.

Тут int не відіграє ролі аргументу і не означає ціле число. Це просто сигнал для компілятора, щоб використовувалася версія постфіксної операції. Для реалізації постфіксного запису операції ++ використовуємо нову функцію:

```
Counter operator++(int);
```

У рядку цієї програми return Counter (count); відбуваються ті самі дії:

```
   Counter operator++(int)
    {
count++;
      return Counter(count);
    }
```

Перевантаження бінарних операцій

Наприклад, якщо A та B – екземпляри класу, для якого перевантажена операція “+” з використанням глобальної або дружньої функції, то вираз A+B трактується як виклик операції-функції: operator +(A,B).

Якщо операція-функція оформлена як компонентна функція, той самий вираз трактується як A.operator+(B).

```
class Counter
{….
Couter operator+(Couter c1)
{
int n=count+c1.count;
return Counter(n);
}

int operator+(Couter c1)
{
int n=count+c1.count;
return n;
}
```

```
 Couter operator+(int f)
 {
 int n=count+f;
 return Counter(n);
 }

 …
};

int main()
{
Counter a, b, c;
c=a+b;
int k=a+b;
Counter q=a+4;
// Counter q=4+a; --
}
```

Існує правило: об'єкт, що стоїть з лівого боку операції (у даному випадку А), викликає функцію перевантаження. Об'єкт, що стоїть праворуч від знаку операції повинен бути переданий в функцію, як аргумент. Операція повертає значення, яке потім використовуємо для своїх потреб.

У функції operator+() до лівого операнду є прямий доступ, оскільки це об’єкт, що викликає функцію. До правого операнду немає доступу, як до аргументу функції.

Інші правила перевантаження операцій

Як відомо, у мові С++ є такі операції над базовими типами даних мови, які є комбінаціями інших операцій. Наприклад, операція *= є комбінацією операцій помножити та присвоїти. Якщо операції = і * перевантажені, то це не означає, ще перевантажено операцію *= (помножити і присвоїти): вона повинна перевантажуватися окремою операцією-функцією (компонентною функцією класу).

Перевантаження стандартної операції для будь-якого класу не змінює правил і змісту виконання цієї операції для виразів, в які не входять екземпляри класу. Наприклад, правила обчислення виразу A+5.2 для float A не змінюються, і змінити їх неможливо. З іншого боку, якщо А є об’єктом класу, для якого виконано перевантаження операції +, то виконується перевантажена операція, якщо вона передбачена для такого випадку. (A.operator + (5.2)).

Припустимо, що А та В є екземпляри класу, для якого перевантажена бінарна операція +. У цьому випадку для обчислення виразів A+B операція-функція може бути оформлена як компонентною функцією, так і глобальною, можливо, дружньою, функцією. Те саме можна сказати і про випадок обчислення виразів типу A+5.

Якщо ж передбачається можливість використання виразів на кшталт 5+A, то операція-функція не може бути оформлена як компонентна функція з тієї причини, що обчислення виразу 5+A еквівалентне виразу 5.operator+(A), що є помилковим. Отже, для цього випадку необхідно визначати операцію-функцію як глобальну чи дружню.

При навантаженні бінарних операцій необхідно враховувати всі можливі поєднання типів операндів. Наприклад, при перевантаженні операції додавання комплексних чисел необхідно враховувати (і це повинен зробити програміст) такі можливі випадки:

- додавання двох комплексних чисел;
- додавання комплексного числа з дійсним числом;
- додавання дійсного числа з комплексним.

На щастя, завдяки угодам про перетворення стандартних типів параметрів можна не враховувати той факт, що додане число може мати ще й довільний числовий тип.

Важливо розуміти, до яких об'єктів відноситимуться аргументи та значення, що повертаються. Коли компілятор зустрічає вираз, він шукає відповідні типи аргументів.

Узагальнюючи викладене вище, можна сказати, що для перевантаження операції завжди потрібна кількість аргументів на одну менше, ніж кількість операндів, оскільки один із операндів є об'єктом, що викликає функцію. Тому для унарних операцій не потрібні аргументи.

Приклад 3:

```
#include <iostream>
using namespace std;
#include <conio.h>
class CPoint
{
protected:
  int x,y;
public:
  CPoint(): x(0), y(0){}
  CPoint(int xx, int yy): x(xx), y(yy) {}
  void get_point(int, int);
  void view_point();
  CPoint operator++();  //1
  CPoint operator++(int); //2
  CPoint operator+(CPoint); //3
  void operator+=(CPoint); //4
};

void CPoint::get_point(int x1, int y1) {
  //cout<<"Enter point"<<endl;
  x=x1;
```

```
  y=y1;
}
void CPoint::view_point(){
cout<<x<<'\n'<<y<<'\n';
}
/*CPoint CPoint ::operator++()
{
  CPoint p1;
  ++x; ++y;
  p1.x=x;
  p1.y=y;
  return p1;
}*/
CPoint CPoint ::operator++()
{
  ++x; ++y;
  return CPoint(x, y);
}
CPoint CPoint ::operator++(int)
{
  CPoint p1;
  p1.x=x++;
  p1.y=y++;
  return p1;
}
CPoint CPoint ::operator+(CPoint p1)
{
  int xx=x+p1.x;
  int yy=y+p1.y;
  return CPoint(xx, yy);
}
void CPoint ::operator+=(CPoint p1)
{
  x+=p1.x;
  y+=p1.y;
}

class CcoloredPoint:public CPoint
{
protected:
  int clcolor;
public:
CcoloredPoint():CPoint(),clcolor(1) {}
  void view_point() {
```

```
  CPoint::view_point();
  cout<<clcolor<<'\n';
  }
void get_point(int xx, int yy, int color){
  CPoint::get_point(xx, yy);
  clcolor=color;
}
};

class CLine: public CPoint
{
protected:
int       x2, y2;
public:
CLine(): CPoint(), x2(1), y2(1) {}
void view_point()
  {
cout<<"Point 1 for line"<<endl;
  CPoint ::view_point();
  cout<<"Point 2 for line"<<endl;
  cout<<x2<<'\n'<<y2<<'\n';
  }
void get_point(int xx1, int yy1, int xx2, int yy2)
  {
  CPoint::get_point(xx1, yy1);
      x2=xx2;
  y2=yy2;
  }
  CLine operator++()
  {       CLine L1;
    CPoint::operator++();
    ++x2; ++y2;
    L1.x=x;
    L1.y=y;
    L1.x2=x2;
    L1.y2=y2;
    return L1;
  }
  void operator+=(CLine L1)
{
CPoint::operator+=(CPoint(L1.x, L1.y));
x2+=L1.x2;
  y2+=L1.y2;
}
```

```
};

int main(){
CPoint point1, point2, point3;
CcoloredPoint colPoint1;
CLine line1, line2, line3;
int x_point, y_point, color_point, x_point2, y_point2;
cout<<" Enter point"<<endl;
cin>>x_point>>y_point;
point1.get_point(x_point, y_point);
point1.view_point();
++point1;
point1++;
point2=++point1;
point1.view_point();
point2++;
cout<<" Show point1"<<endl;
point1.view_point();
cout<<" Show point2"<<endl;
point2.view_point();
point1+=point2;
cout<<" Show point1+=point2"<<endl;
point1.view_point();
point3=point1+point2;
cout<<" Show point3"<<endl;
point3.view_point();

cout<<'\n'<<"CcoloredPoint"<<endl;
cin>>x_point>>y_point>>color_point;
colPoint1.get_point(x_point, y_point, color_point);
cout<<"Show CcoloredPoint"<<endl;
colPoint1.view_point();
cout<<"pref ++"<<endl;
++colPoint1;
colPoint1.view_point();

cout<<" Enter point1 for line"<<endl;
cin>>x_point>>y_point;
cout<<" Enter point2 for line"<<endl;
cin>>x_point2>>y_point2;
line1.get_point(x_point, y_point, x_point2, y_point2);
cout<<" Points line1"<<endl;
line1.view_point();
++line1;
```

```
cout<<" ++line1"<<endl;
line1.view_point();
cout<<" Show line1+=line2"<<endl;
line1+=line2;
line1.view_point();
cout<<" Show line3++"<<endl;
line3++;
line3.view_point();
_getch();
return 0;
}
```

Лекція 12. НАСЛІДУВАННЯ (УСПАДКУВАННЯ)

Наслідування – це можливість створення ієрархії класів, коли нащадки успадковують усі властивості своїх предків, можуть їх змінювати та додавати нові.

Ієрархія класів є деревовидною. Класи, що знаходяться ближче до початку ієрархії, поєднують у собі загальні риси. У міру просування ієрархією вниз класи набувають все більш конкретних рис. У С++ кожен клас може мати скільки завгодно нащадків і предків. Існуючий клас називається базовим чи батьківським класом, а побудовані на його основі класи – похідними, і навіть нащадками чи спадкоємцями. Кожен із похідних класів може, своєю чергою, виступати у ролі базового класу.

Необхідність у наслідуванні виникає, оскільки великою кількістю не пов'язаних між собою класів складно керувати.

При наслідуванні базовий клас залишається незмінним.

Просте успадкування – за якого похідний клас має одного з батьків.

Множинне успадкування – за якого похідний клас може мати більше одного з батьків.

Похідний клас отримує доступ до всіх полів і методів базового класу відповідно до специфікатора доступу, але при цьому поля та методи базового класу не копіюються, а залишаються його приналежністю.

Специфікатор доступу protected

Для будь-якого елемента класу може також використовуватися специфікатор доступу protected, який для одиночних класів, що не входять до ієрархії, є рівносильним private. Різниця між private та protected виявляється у наслідуванні. У таблиці 7 наведено можливості використання специфікаторів доступу.

Таблиця 7

Можливості використання специфікаторів доступу

Специфікатор доступу для елементів базового класу	Доступ у самому класі (базовому)	Доступ із похідних класів	Доступ із зовнішніх класів та функцій (у тому числі main())
public	+	+	+
protected	+	+	-
private	+	-	-

Формат опису похідного класу

У випадку визначення похідного класу він має такий формат:

class ім'я_похідного_класу : ключ_доступу
ім'я_базового_класу {тіло_похідного_класу};

Ключ доступу задається, як і при вказанні доступу до полів та методів класу, за допомогою тих самих специфікаторів public, private та protected.

Приклад 1:

```
class A{/* тіло_класу A*/};
class B: public A{/* тіло_класу B*/};
class C{/* тіло_класу C*/};
class D: private A, public B {/* тіло_класу D*/}; – множинне наслідування
```

Можливість звернення до елементів базових класів регулюється ключами доступу, які наведені в таблиці 8.

Таблиця 8

Вплив ключів доступу при наслідуванні

Ключ доступу (при наслідуванні)	**Специфікатор доступу в базовому класі**	**Доступ у похідному класі**
private	private	-
	protected	private
	public	private
protected	private	-
	protected	protected
	public	protected
public	private	-
	protected	protected
	public	public

Приклад 2:

```
class AA {
private:
int dataA1;

protected:
int dataA2;

public:
int dataA3;};

class BB: public AA
{ public:

int dataB;

void func(){
int a;
```

```
a=dataA1; // error
a=dataA2;
a=dataA3;
}};

int    main(){
int  k;
BB  objB;
k=objB.dataA1; //error
k=objB.dataA2; //error
k=objB.dataA3;
return 0;
}
```

Якщо і в базовому, і в похідному класах існує метод з однією і тією ж сигнатурою, то при виклику його з об'єктом похідного класу буде виконано метод похідного класу. І тут кажуть, що метод похідного класу перевантажує метод базового класу. Об'єктам базового класу нічого не відомо про похідний клас. Тому завжди використовують методи базового класу.

Уніфікована мова моделювання (Unified Modeling Language, UML)

Сьогодні процес створення складних програмних систем неможливо уявити без поділу на етапи життєвого циклу. Під життєвим циклом програми розумітимемо сукупність етапів:

- аналіз предметної галузі та створення ТЗ (при взаємодії із замовником);
- проєктування структури програми;
- кодування (набір програмного коду згідно з проєктною документацією);
- тестування та налагодження;
- використання програми;
- супровід програми;
- утилізація.

Зупинимося більш докладно на процесі проєктування. Під час проєктування архітектором чи досвідченим програмістом створюється проєктна документація, куди входять текстові описи, діаграми, моделі майбутньої програми. У цьому процесі бере участь мова UML.

UML – мова графічного опису в галузі розроблення програмного забезпечення, моделювання бізнес-процесів, системного проєктування та відображення організаційних структур.

UML є мовою широкого профілю, це відкритий стандарт, що використовує графічні позначення для створення абстрактної моделі системи, що називається UML-моделлю. UML не є мовою програмування.

UML дозволяє створювати різні діаграми, які відображають різні погляди на програму та її дії з різних точок зору.

Деякі види діаграм UML

Структурні діаграми:

- діаграма класів;
- діаграма компонентів (Component diagram) – статична структурна діаграма, що показує розбиття програмної системи на структурні компоненти та зв'язки (залежності) між компонентами. Як фізичні компоненти можуть виступати файли, бібліотеки, модулі, виконувані файли, пакети тощо;
- діаграма композитної / складової структури (діаграма кооперації);
- діаграма об'єктів;
- діаграма прецедентів.

Діаграми поведінки:

- діаграма діяльності;
- діаграма станів;
- діаграма варіантів використання.

Діаграми взаємодії:

- діаграма послідовності;
- діаграма синхронізації і т. д.

Приклад. Множинне наслідування:

```
//файл student.h
class student
{
protected:
  char education[15];
  char degree[10];
public:
  student();
  student(char*, char*);
  ~student();
  void getedu();
  void putedu();
};

//файл student.cpp
#include "student.h"
```

```
#include "stdlib.h"
#include <iostream>
using namespace std;
student::student(){
  strcpy(education,"KhAI");
  strcpy(degree,"master");
}

student::student(char* edu, char* deg)
{
  strcpy(education,edu);
  strcpy(degree,deg);
}

void student::getedu()
{cout<<"Enter education"<<endl;
 cin>>education;
 cout<<"Enter degree"<<endl;
 cin>>degree;
}

student::~student()
{}

void student::putedu()
{
cout<<"Education: "<<education<<endl;
cout<<"Degree: "<<degree<<endl;
}

//файл employee.h
class employee
{
protected:
char surname[20];
public:
  employee();
  employee(char *);
  ~employee();
  void getdata();
  void putdata();
};

// файл employee.cpp
```

```
#include "employee.h"
#include "stdlib.h"
#include <iostream>
using namespace std;

employee::employee()
{
 strcpy(surname,"Popov");
}

employee::employee(char * sur)
{
 strcpy(surname, sur);
}

void employee::getdata()
{
cout<<"Enter surname"<<endl;
 cin>>surname;
}
void employee::putdata()
{
cout<<"Surname: "<<surname<<endl;
}

employee::~employee()
{
}

// файл мн.cpp :
#include <stdlib.h>
#include <conio.h>
#include <iostream>
#include "employee.h"
#include "student.h"
using namespace std;
class manager: public student, public employee
{
private:
  char job[15];
  float salary;
public:
manager():student(), employee(), salary (0.0) {strcpy(job,"engineer");}
```

```
manager(char * ed1, char* d1, char * name1, char* job1, float s1)
:student(ed1, d1), employee(name1), salary(s1) {strcpy(job,job1);}

  void getdata()
  {
employee::getdata();
student::getedu();
cout<<"Enter job"<<endl;
cin>>job;
cout<<"Enter salary"<<endl;
cin>>salary;
  }
void putdata()
  {
employee::putdata();
student::putedu();
cout<<"Job: "<<job<<endl;
cout<<"Salary: "<<salary<<endl;
  }
};

int _tmain(int argc, _TCHAR* argv[]) {
  student s1(“fghfg”,”uioip”), s2;
manager person1, person2("KAI", "stylist","Tarasenko D.", "developer",
15000);
  person1.getdata();
  person2.putdata();
  cout<<"----------------------"<<endl;
  person1.putdata();
_getch();
return 0;}
```

На основі написаного коду в середовищі Visual Studio була автоматично побудована UML-діаграма класів, яка представлена на рис. 9.

UML – універсальна мова моделювання, що призначена для моделювання програм. Під моделюванням розуміємо створення наочної віртуальної інтерпретації чого-небудь. UML дозволяє створювати подібну інтерпретацію програм високорівневої організації.

Найбільш важливим засобом UML є набір різних видів діаграм. Діаграми класів демонструють відносини між різними класами. Діаграми класів відображають погляди на програму і її дії з різних точок зору.

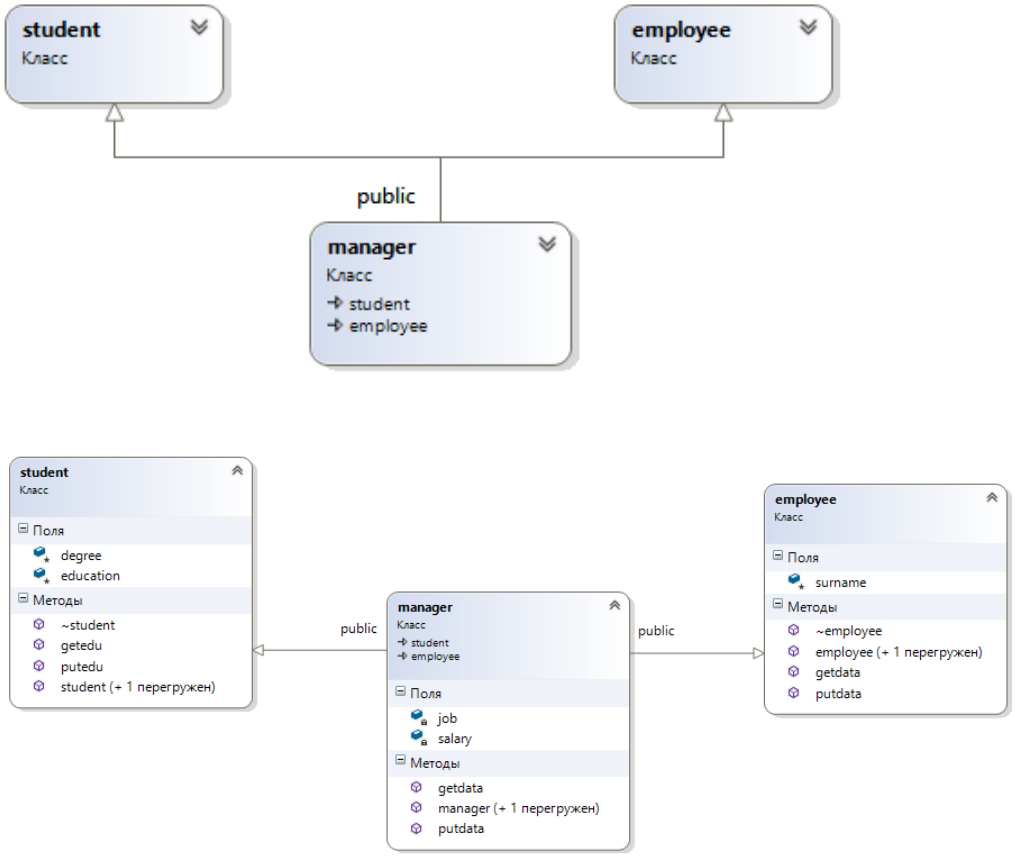

Рис. 9. UML-діаграма класів

Лекція 13. ВІРТУАЛЬНІ ФУНКЦІЇ. ПОЛІМОРФІЗМ

Поліморфізм – можливість використовувати у різних класах ієрархії одне ім'я функції для визначення схожих дій і гнучко вибирати необхідну дію під час виконання програми. Часто поняття поліморфізму асоціюється з механізмом роботи віртуальних методів.

Приклад 1:

```
class Base {
public:
void Show() {cout<<"Base \n" ; }
};

class Next1 : public Base {
public:
void Show() {cout<<"Next1 \n" ; }
};

class Next2 : public Base {
public:
void Show() {cout<<"Next2 \n" ; }
};

int main () {
   Next1 N1;
   Next2 N2;
   Base *p; //вказівник на базовий клас
   p = &N1;
   p -> Show(); //  Base
   p = &N2;
   p -> Show(); //  Base

   return 0;
}
```

Якщо один і той самий метод існує і в базовому, в похідному класі, то для об'єктів похідного класу буде виконуватися метод похідного класу. Об'єктам базового класу нічого невідомо про похідний клас, тому вони завжди використовують методи базового класу.

Приклад 2:

```
class Base {
public:
virtual void Show() {cout<<"Base \n" ; }
};

class Next1 : public Base {
```

```
public:
void Show() {cout<<”Next1 \n” ; }
};

class Next2 : public Base {
public:
void Show() {cout<<”Next2 \n” ; }
};

int main () {
   Next1 N1;
   Next2 N2;
   Base *p; //вказівник на базовий клас
   p = &N1;
   p -> Show(); //  Next1
   p = &N2;
   p -> Show(); //  Next2
   return 0;
}
```

Якщо в базовому класі метод визначений як віртуальний, метод визначений у похідному класі з тим самим ім'ям і набором параметрів, автоматично стає віртуальним, а з відмінним набором параметрів – звичайним.

Різні методи можуть виконуватися за допомогою одного і того самого виклику. Це і є сутність поліморфізму: один і той самий рядок коду може викликати різні методи.

Приклад 3:

```
class Shape {
public:
virtual void drow(){ … }
};
class triangle : public Shape {
public:
void drow(){ … }
};
class rect : public Shape {
public:
void drow(){ … }
};
class circle : public Shape {
public:
void drow(){ … }
};
```

```
int main () {
  Shape *ptarr [100];
   triangle T1;
   rect R1;
   circle C1;
ptarr[0]= & T1;
ptarr[1]= & R1;
ptarr[2]= & C1;
…
   for (int j = 0; j < 100; j++)
   ptarr[j] -> drow();
}
```

На рис. 10 наведено ієрархію класів у прикладі 3.

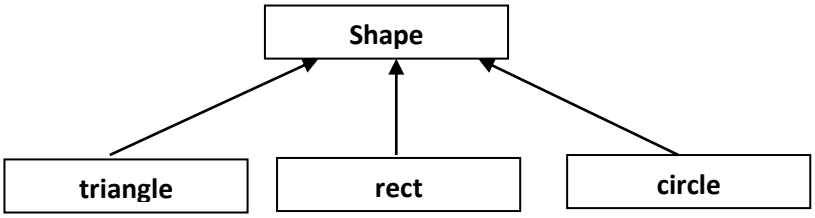

Рис. 10. Ієрархія класів у прикладі 3

Щоб використовувати поліморфізм, необхідно виконати умову:

- всі класи повинні бути похідними одного і того ж класу;
- функція drow повинна бути оголошена віртуальною (virtual) в базовому класі.

Робота з об'єктами найчастіше проводиться через покажчики. Покажчику на базовий клас можна присвоїти значення адреси будь-якого похідного класу. У різний час об'єкт може посилатися на об'єкти різних класів ієрархії, але можна використовувати явне перетворення типу даних. Поліморфізм – одна з ключових властивостей ООП.

Абстрактні класи

Абстрактним класом називається клас, об'єкти якого ніколи не будуть реалізовані (він є ланкою для створення ієрархічної структури). У цих класів немає об'єктів.

Ознака абстрактного класу: присутність в описі класу хоча б однієї чисто віртуальної функції.

Чисто віртуальна функція – це функція, після оголошення якої додано вираз = 0. Таким чином, якщо клас використовує чисто віртуальну функцію, він стає абстрактним. Жодні об'єкти з нього реалізувати не вдасться.

Визначення чисто віртуальної функції в базовому класі не є обов'язковим.

Приклад:

```
#include <stdlib.h>
#include <conio.h>
#include <iostream>
using namespace std;
class person
{
   protected:
   char name[20];
   public:
   person() {strcpy(name,"Ivan");}
   person(char * n) {strcpy(name,n);}
   void getName(){ cout<<"Enter name"<<endl;
                cin>>name;}
   void putName() {cout<<"Name: "<<name<<endl;}
   virtual void getData()=0;
   virtual char * func_if()=0;
};

class student : public person
{
  private:
  float sr_b;
  public:
  student(): person(), sr_b(0.0){}
  student(char * name_stud, float s): person(name_stud), sr_b(s) {}
  void getData(){
  person::getName();
  cout<<"Enter avg. mark"<<endl;
  cin>>sr_b;
  }
  char * func_if(){
  if (sr_b>=4)  return "The student receives a scholarship."; else return "The
student does not receive a scholarship."; }
};

class prepod : public person
{
  private:
  int pub;
  public:
  prepod(): person(), pub(0){}
  prepod(char * name_prepod, int p): person(name_prepod), pub(p) {}
  void getData(){
```

```
  person::getName();
  cout<<"Enter publ."<<endl;
  cin>>pub;
  }
  char * func_if(){
  if (pub>=20)  return "Complies!"; else return "Does not the requirements.";
}
};

int main() {
//person man1;
prepod pr1("Petrov", 25);
student st1("Maksimenko", 3.5);
/*person *p;
p=&st1;
p->putName();
cout<<p->func_if()<<endl;
p=&pr1;
p->putName();
cout<<p->func_if()<<endl;*/
person *p[2];
p[0]=&st1;
p[1]=&pr1;
for (int i=0; i<2; i++)
{p[i]->putName();
cout<<p[i]->func_if()<<endl;}
_getch();
return 0;
}
```

Лекція 14. ШАБЛОНИ КЛАСІВ

Бібліотека стандартних шаблонів (Standard Template Library, STL) – стандартна бібліотека шаблонів.

Сутність в STL – контейнер, спосіб організації зберігання даних (приклад контейнера: масив, стек, пов'язаний список, черга).

Алгоритми в STL – передбачає процедури, що застосовуються до контейнерів для обробки їх даних різними способами.

Ітератори в STL – нагадують покажчики для доступу до окремих даних.

Шаблони класів

Шаблони класів відрізняються від шаблонів функцій способом реалізації.

Для створення шаблонної функції необхідно викликати її з аргументами потрібного типу. Класи ж реалізуються за допомогою визначення об'єкта, що використовує шаблонний аргумент:

Stack<float> s1; // Stack із STL (стандартна бібліотека шаблонів).

Такий вираз створить об'єкт s1. Це стек, у якому зберігаються числа типу float. Компілятор резервує область пам'яті для даних цього об'єкта, використовуючи тип float скрізь, де з'являється аргумент Type у специфікації класу. Резервується місце і під методи класу. Звичайно ж, методи використовують тільки тип float.

Створення об'єкта типу Stack, що зберігає об'єкти інших типів, як у виразі: Stack<long> s2; означає не тільки резервування іншого об'єму у пам'яті для даних, але і створення нового набору методів, що оперують типом long.

Приклад:

```
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
#include <conio.h>
#include <ctime>
using namespace std;
template <class TypeElem>
class array_class{
public:
  array_class();
  array_class(int size_array);
  ~array_class();
  void addElem(TypeElem);
  bool delElem(int i);
  void showArray();
  float avg();
```

```
  TypeElem sum();
  TypeElem min();
  void operator+=(TypeElem);
  TypeElem & operator [] (int);
  bool SetCapacity(int NewCapecity);
private:
  TypeElem * data;
  int size, index;};
template <class TypeElem> array_class<TypeElem>::array_class(){
  data=0;
  index=0;
  size=0;}
template <class TypeElem> array_class<TypeElem>::array_class(int
size_array){
  size=size_array;
  index=0;
  data=new TypeElem [size_array];}
template <class TypeElem> array_class<TypeElem>::~array_class(){
  if (data) delete [] data;}
template <class TypeElem>
 void array_class<TypeElem>::addElem(TypeElem value){
if (index==size) cout<<"Error addElem"<<endl;
else {data[index]=value;
   index++;}}
template <class TypeElem>
 bool array_class<TypeElem>::delElem(int i){
  if (size < 0 || index >= size) return false;
  for (int nom = i; nom < size; nom++)
        data[nom] = data[nom + 1];
  size--;
  return true;}
template <class TypeElem>
 void array_class<TypeElem>::showArray(){
  cout<<"Show array\n";
for (int i=0; i<size; i++)
  cout<<data[i]<<endl;}
template <class TypeElem> float array_class<TypeElem>::avg(){
  TypeElem sum=0;
  for (int i=0; i<size; i++)
        sum+=data[i];
  return (sum/size);}
template <class TypeElem>
 TypeElem array_class<TypeElem>::sum(){
  TypeElem s=0;
```

```
  for (int i=0; i<size; i++)
        s+=data[i];
  return s;}
template <class TypeElem> TypeElem
array_class<TypeElem>::min(){
  TypeElem min=data[0];
  for (int i=0; i<size; i++)
        if (min>data[i]) min=data[i];
return  min;}
template <class TypeElem>
 void array_class<TypeElem>::operator+=(TypeElem value){
for (int i=0; i<size; i++)
  data[i]+=value;
}
template <class TypeElem>
TypeElem & array_class<TypeElem>::operator[](int nomElem){
if ((nomElem>=0) && (nomElem<size)) return data[nomElem];}
template <class TypeElem>
 bool array_class <TypeElem>::SetCapacity(int NewCapacity){
  if (NewCapacity<0) return false;
  int OldCapacity=size;
  size=NewCapacity;
  TypeElem*temp=new TypeElem [size];
  if (NewCapacity>OldCapacity)
  {
  for (int i=0; i<OldCapacity; i++)
        temp[i]=data[i];
  for (int i=OldCapacity; i<NewCapacity; i++)
        temp[i]=0;
  }
  if (NewCapacity<=OldCapacity)
        for (int i=0; i<NewCapacity; i++)
        temp[i]=data[i];
  delete [] data;
  data=temp;
  return true;}
int _tmain(int argc, _TCHAR* argv[])
{int n;
  cout<<"Enter n"<<endl;
  cin>>n;
  array_class <int> arr(n);
  srand(time(0));
  for (int i=0; i<n; i++)
  arr.addElem(rand()%100-rand()%100);
```

```
arr.showArray();
cout<<"sum= "<<arr.sum()<<endl;
cout<<"avg= "<<arr.avg()<<endl;
cout<<"min= "<<arr.min()<<endl;
int k=3;
arr+=k;
arr.showArray();
arr[2]=19;
cout<<"Elem №2= "<<arr[2]<<endl;
arr.SetCapacity(7);
arr.showArray();
arr.delElem(1);
arr.showArray();
_getch();
return 0;}
```

Лекція 15. ДРУЖНІ ФУНКЦІЇ

Дружні функції є між класами. Дружньою (класу) називається така функція, яка має доступ до всіх його компонентів, у тому числі приватних (private) і захищених (protected), і при цьому не є компонентною функцією цього класу. Насамперед зазначимо, що функція не може стати дружньою "без відома" самого класу: вона має бути оголошена в ньому зі специфікатором friend.

Приклад 1:

```
class Counter
   {
   private:
     int count;
   public:
     Counter() : count(5) { }          // конструктор без параметрів
     Counter(int c) : count(c) { }          // конструктор з параметром

     friend Counter operator+(Counter c, int n) ;// дружня функція
     friend Counter operator+( int n, Counter c);
   };

  Counter operator+(Counter c, int n) // визначення функції
{
   return Counter (c.count + n);
 }

friend Counter operator+( int n, Counter c)
{
   return Counter (n+ c.count);
 }

int main(){

 Counter c1, c2;
Counter c3=c1+15;
Counter c4=30+c2;
}
```

Дружня функція розглядатиме об'єкти двох класів як аргументи та оброблятиме їх приховані дані.

Приклад 2:

```
#include <iostream>
using namespace std;
class beta;             // потрібно для оголошення frifunc
```

```
class alpha
   {
   private:
     int data;
   public:
     alpha() : data(3) {  }          // конструктор без параметрів

     friend int frifunc(alpha, beta); // дружня функція
   };

class beta
   {
   private:
     int data;
   public:
     beta() : data(7) {  }           // конструктор без параметрів

     friend int frifunc(alpha, beta); // дружня функція
   };

int frifunc(alpha a, beta b)          // визначення функції
{
   return (a.data + b.data);
 }

int main()
{
   alpha aa;
   beta bb;

   cout << frifunc(aa, bb) << endl;   // виклик функції
   return 0;
}
```

Необхідно, щоб функція frifunc() мала доступ до прихованих даних обох класів, тому вона є дружньою функцією. Для такої реалізації в оголошеннях усередині кожного класу використовується ключове слово friend:

```
friend int frifunc(alpha, beta);
```

Це оголошення може бути розташоване будь-де всередині класу. Немає жодної різниці, буде записано воно в public або private секції. Об'єкт кожного класу передається як параметр функції frifunc(), і функція має доступ до прихованих даних обох класів за допомогою цих аргументів.

З одного боку, дружні функції підвищують гнучкість мови, але, з іншого боку, вони не відповідають принципу обмеження доступу до даних, що свідчить, про те, що тільки функції-члени можуть мати доступ до прихованих даних класу. Але програміст, який не має доступу до коду класу, не може зробити функцію дружньою. У цьому сенсі цілісність даних зберігається.

Угоди щодо дружніх функцій такі:

1. Дружня функція має обов'язково отримувати екземпляр класу як параметр.

2. Дружня функція має доступ до всіх полів класу, у тому числі приватних і захищених, за допомогою уточнених (кваліфікованих) імен.

3. Оскільки дружня функція не є компонентною функцією, вона не має безпосереднього доступу до компонентних даних, навіть якщо вони оголошені як загальнодоступні.

4. Місце розташування прототипу або визначення функції в класі не має значення, і права доступу функції до компонентних даних не залежать від специфікаторів доступу, оскільки дружня функція не є компонентною функцією класу.

Дружні функції можна використовувати у випадках, коли потрібно виконати спільну обробку даних кількох різних класів. Крім того, вони знаходять своє застосування при перевантаженні операцій.

Дружні класи

Методи можуть бути перетворені на дружні функції одночасно з визначенням всього класу як дружнього.

Для класу alpha весь клас beta проголошений дружнім. Тепер усі методи beta мають доступ до прихованих даних класу alpha:

```
// дружні класи
#include <iostream>
using namespace std;

class alpha
  {
  private:
    int data1;
  public:
    alpha() : data1(99) {  }  // конструктор
    friend class beta;        // beta – дружній клас
  };

class beta
  {                         // всі методи мають доступ
  public:                   // до прихованих даних alpha
    void func1(alpha a)  { cout << "\ndata1 =" << a.data1; }
```

```
};

int main()
{
  alpha a;
  beta b;
  b.func1(a);
  cout << endl;
  return 0;
}
```

Можна було б зробити і так: спочатку оголосити звичайний клас beta:

class beta;,

потім оголосити клас alpha, а всередині визначення alpha оголосити дружність класу beta:

friend beta;.

БІБЛІОГРАФІЧНІЙ СПИСОК

1. Hunter G. Using Embedded Template Library (ETL) with C++. URL: https://blog.mbedded.ninja/programming/languages/c-plus-plus/cpp-on-embedded-systems/

2. Maier A., Sharp A., Vagapov Y. Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things // In Proceedings of the 2017 Internet Technologies and Applications (ITA). IEEE. 2017. P. 143–148.

3. Plauska I., Liutkevičius A., Janavičiūtė A. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics.* 2023. Vol. 12 (1). DOI: 10.3390/electronics12010143

4. Grimes R. Beginning C++ Programming. UK : Packt Publishing Ltd., 2017. 516 p.

5. Васильєв О. М. Програмування С++ в прикладах і задачах : навч. посіб. Київ : Ліра-К, 2019. 382 с.

6. Трофименко О. Г., Прокоп Ю. В., Логінова Н. І., Задерейко О. В. С++. Алгоритмізація та програмування : підручник. Одеса : Фенікс, 2019. 477 с.

ЗМІСТ

Навчальне видання

Лутай Людмила Миколаївна
Нікітін Артем Олексійович
Бурдейна Вікторія Михайлівна

СТВОРЕННЯ ДОДАТКІВ ДЛЯ ПРОГРАМНО-АПАРАТНИХ КОМПЛЕКСІВ

КОНСПЕКТ ЛЕКЦІЙ

Редактор Костіна І.В.

Зв. план, 2025

Підписано до видання 07.03.2025
Ум. друк. арк. 6,9. Обл.-вид. арк. 7,75. Електронний ресурс

Видавництво «Факт»
Україна, 61166, м. Харків, вул. Бакуліна, 11, оф. 2-26.
+38 (050) 323-22-01, publish_fakt@ukr.net
Свідоцтво суб'єкта видавничої справи ДК № 3172 від 22.04.2008