**Лекція 12. Конструктори. Ініціалізація даних**

**Конструктори і деструктори.**

Серед функцій-членів класу можуть бути такі, що визначають процеси створення, ініціювання, копіювання та знищення об’єктів свого класу. До цих функцій належать конструктори і деструктори, які, якщо явно не оголошуються, то автоматично викликаються системою за замовчуванням.

Чим і як може ініціалізуватися клас?

* Деякими замовчуваними значеннями
* Списком значень, що відповідає типам даних
* Об’єктом цього ж класу визначеним раніше
* Деякими даними сумісними за типом з типом класу

У мові С++ для ініціалізації об'єктів призначений механізм, називаний ***конструктором***. Це — функція-член, ім'я якої збігається з ім'ям класу. Вона може мати будь-які параметри, що необхідні для ініціалізації полів об'єкта. При цьому конструктор не має ніякого значення, що повертається. Його виклик являє собою визначення об'єкта. Інакше кажучи, визначаючи об'єкт, ви викликаєте конструктор.

**Види конструкторів:**

* За замовчуванням
* Ініціалізації (з параметрами)
* Копіювання
* Додаткові конструктори

Головною метою **конструкторів** є ініціювання змінних-об’єктів класу та розподілення пам’яті для їх зберігання. Конструктор викликається кожного разу при створенні об’єкта даного класу або явно, або автоматично.

**Дії, які виконують конструктори.**

* Виділяють динамічну пам’ять
* Ініціалізують дані класу
* Якщо програміст не вказав жодного конструктора або якісь поля не були проініціалізовані, то полям значимих типів присвоюється нуль, полям показівникових типів — значення null;
* Не можуть нічого повернути
* Забороняють або дозволяють і виконують приведення до типу класу
* Можуть викликатися явно та неявно

**Основні правила роботи з конструкторами**:

* ім’я конструктора повинне співпадати з ім’ям свого класу;
* для конструктора не вказується тип значення, яке повертає функція;
* клас може мати декілька конструкторів або не мати жодного;
* конструктор за замовчуванням — це конструктор, який не має параметрів, або всі його параметри мають значення за замовчуванням;
* конструктор копіювання безпосередньо призначений для створення об’єкта класу шляхом копіювання даних з існуючого об’єкта.

**Конструктор за замовчуванням** – це конструктор, який дозволяє створювати екземпляри класів з неявною ініціалізацією даних**.** Такий конструктор автоматично створюється компілятором для класів, в яких не визначений власний конструктор. Проте, як тільки в класі визначається хоч один конструктор, такий автоматичний конструктор перестає діяти. В разі необхідності створення екземплярів без ініціалізації, варто визначити в класі конструктор за умовчанням. Найпростіший спосіб зробити це – перевантажити конструктор, або визначити умовчання для всіх його параметрів.

**Конструктор ініціалізації** містить окремі значення, що використовуються для ініціалізації стану полів екземпляру класу. У списку параметрів може бути зазначений нуль, один чи більше параметрів будь-якого типу. В прикладі минулої лекції це

Point (double x=0, double y=0): \_x(x),\_y(y) { };

Це скорочена форма запису, конструктор може мати вигляд

Point (double x, double y){ \_x=x; \_y=y; };

Приклад минулої лекції

#include <iostream>

using namespace std;

class Point

{// Атрибути -private закрили прямий доступ до атрибутів

double \_x, \_y;

int \*ptr; //\* Покажчик на ділянку пам‘яті

// відкрили доступ до методів

public: // Конструктор

Point (double x=0, double y=0**): \_x(x),\_y(y) { }**;

Point (const Point &obj) // Конструктор копіювання

{ cout << "\nCopy Constr\n"; }

~Point(){cout << "\nDestruct\n";}; // Деструктор

// Функції доступу до атрибутів - селектори

double& x() {return \_x;}

/\* ім'я \_х в тексті функції позначає поле \_х того екземпляру, до якого застосовано функцію \*/

double& y() {return \_y;}

void set\_x(double a){\_x=a;}

void set\_y(double a){\_y=a;}

};

void funcShow(Point object)

{ cout << "\nFunction get object as param\n"; }

Point funcReturnObject()

{ Point object;

cout << "\nFunction return object\n";

return object;

}

int main(int argc, char\* argv[])

{ system("color F0");

Point a1(1,2);

cout<< a1.x() <<";"<< a1.y() <<endl;

Point b1(1);

cout<< b1.x() <<";"<< b1.y() <<endl;

Point c1;

cout<< c1.x() <<";"<< c1.y()<<endl;

cout << "1 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

Point obj1; // створюємо об‘єкт класу

cout << "2 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

funcShow(obj1); // передаємо об‘єкт до функції

cout << "3 - 4 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

funcReturnObject(); // функція повертає об‘єкт

cout << "5 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

Point obj2 = obj1; // ініціалізація об‘єкту при створенні

Point \*ptr;

ptr = new Point (22,33);

cout<< ptr->x() <<";"<< ptr->y() <<endl;

delete ptr;

Point \*ptrM = new Point[3];

for (int j=0; j<3; j++)

{ptrM[j].set\_x(j); ptrM[j].set\_y(j);

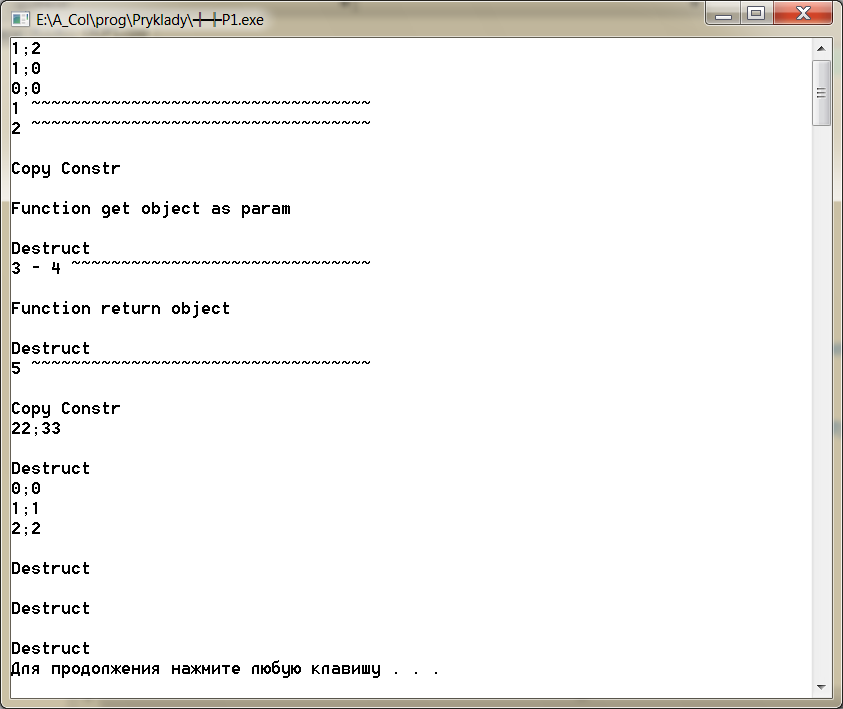
cout<< ptrM[j].x() <<";"<< ptrM[j].y() <<endl; }

delete [] ptrM ;

system("pause");

return 0;

}



Використання списку ініціалізаторів членів **переважніше**, ніж встановлення значень в тілі конструктора, оскільки він безпосередньо ініціалізує елемент. В наданому прикладі показано список ініціалізаторів членів, який складається зі всіх виразів ***ідентифікаторів (аргументів)*** після двокрапки (":"). Ідентифікатор повинен посилатися на член класу; він ініціалізується значенням аргументу. Аргумент може бути одним з параметрів конструктора, викликом функції.

*Список ініціалізації — єдиний спосіб задати значення константних змінних-членів.* Крім того, така ініціалізація буває більш ефективною. Правда, якщо клас містить велику кількість членів, список може стати занадто довгим, що знизить наочність програми.

Можна описати будь-яку кількість конструкторів класу. Вони повинні відрізнятися кількістю або типами параметрів (властивість поліморфізму).

**class** Country

{

**private**:

**char** name[40];

**double** area;

**int** population;

**public**:

Country(**double** area) { **this**->area = area; }

Country(**const** **char**\* name);

Country(**const** **char**\* name, **double** area);

Country(**const** **char**\* name, **double** area, **int** population);

. . .

};

Country::Country(**const** **char** \* name)

{

strcpy(**this**->name, name);

area = 1; // істотно, щоб територія не мала значення 0

// і не виникало помилки у функції density()

}

Country::Country(**const** **char** \* name, **double** area)

{

strcpy(**this**->name, name);

**this**->area = area;

}

Country::Country(**const** **char** \* name, **double** area, **int** population)

{

strcpy(**this**->name, name);

**this**->area = area;

**this**->population = population;

}

**Конструктор копіювання** ініціалізує стан класу значення іншого екземпляру цього класу. В списку параметрів вказується єдиний параметр, що має тип “посилання на екземпляр класу”. Для явного визначення конструктора копіювання необхідно дотримуватись особливого синтаксису:

**<ім’я\_класу> (const <ім’я\_класу> & );**

Параметром конструктора копіювання є стала змінна-посилання на екземпляр класу. Його призначення – коректне створення копії екземпляру. Особливо важливо це у випадку, коли членами класу є вказівники, пам’ять під які виділяється оператором **new**. Адже тоді поелементне копіювання копіює вказівник (поверхневе копіювання), а не об’єкт, на який він посилається, – для цього необхідне глибоке копіювання.

*Конструктор копіювання потрібен нам для того, щоб створювати «реальні» копії об'єктів класу, а не побітову копію об'єкта*. Іноді це принципово важливо. Таку «реальну» копію об'єкта треба створювати в декількох випадках:

* коли ми передаємо об'єкт в яку-небудь функцію у вигляді параметра;
* коли яка-небудь функція повинна повернути об'єкт класу в результаті своєї роботи;
* коли ми в головній функції один об'єкт класу ініціалізуємо іншим об'єктом класу.

Наприклад, ми передаємо об'єкт в функцію у вигляді параметра. Функція буде працювати не з самим переданим об'єктом, а з його побітової копією. Припустимо в конструкторі класу, при створенні об'єкта, виділяється певний об'єм пам'яті, а деструкція класу цю пам'ять звільняє. Покажчик побітової копії об'єкта буде зберігати ту ж адресу пам'яті, що й оригінальний об'єкт. І, коли при завершенні роботи функції і знищенні побітової копії об'єкта, спрацює деструктор, він обов'язково звільнить пам'ять, яка була зайнята об'єктом-оригіналом. На додачу, ще й при завершенні роботи програми, деструктор спрацює повторно і спробує ще раз звільнити цей обсяг пам'яті, що неминуче призведе до помилок програми. Та ж доля спіткає і пам'ять, виділену для покажчика об'єкта, якщо буде віддалятися побітова копія повертається функцією об'єкта, і побітова копія при ініціалізації об'єкту класу іншим об'єктом.

Щоб уникнути цих проблем і помилок існує конструктор копіювання. Його робота полягає в тому, щоб створити реальну копію об'єкта зі своєю особистою виділеної динамічною пам'яттю.

Розглянемо конструктор копіювання.

#include <iostream>

**#include <cassert>**

class Drob

{

private:

    int m\_numerator;

    int m\_denominator;

public:

    // Конструктор по умолчанию

    Drob(int numerator=0, int denominator=1) :

        m\_numerator(numerator), m\_denominator(denominator)

    {

**assert**(denominator != 0);

    }

    // Конструктор копирования

    Drob(const Drob &drob) :

        m\_numerator(drob.m\_numerator), m\_denominator(drob.m\_denominator)

/\* Примечание: Мы имеем прямой доступ к членам объекта drob, поскольку мы сейчас находимся внутри класса Drob \*/

    {

/\*Нет необходимости выполнять проверку denominator здесь, так как эта проверка уже осуществляется в конструкторе класса Drob\*/

        std::cout << "Copy constructor worked here!\n"; /\* просто, чтобы показать, что это работает\*/

    }

*friend std::ostream& operator<<(std::ostream& out, const Drob &d1);*

};

*std::ostream& operator<<(std::ostream& out, const Drob &d1)*

{

out << d1.m\_numerator << "/" << d1.m\_denominator;

return out;

}

int main()

{

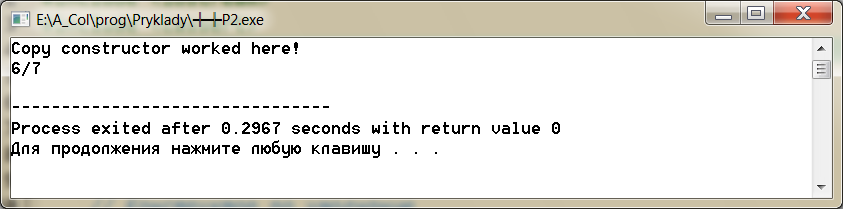
Drob sixSeven(6, 7); /\* прямая инициализация объекта класса Drob, вызывается конструктор Drob(int, int)\*/

Drob dCopy(sixSeven); /\* прямая инициализация, вызывается конструктор копирования класса Drob\*/

std::cout << dCopy << '\n';

}

Результат виконання програми:



Конструктор копіювання в цьому прикладі використовує почленну ініціалізацію та функціонально є еквівалентним конструктору по замовчуванню, за винятком того, що додано оператор виведення, в якому вказано текст (що конструктор копіювання спрацював). **Почленна ініціалізація** означає, що кожен член об’єкта-копії ініціалізується безпосередньо з члена об’єкта-оригіналу. Тобто в прикладі, наведеному вище, dCopy.m\_numerator матиме значення sixSeven.m\_numerator (6), а dCopy.m\_denominator дорівнюватиме sixSeven.m\_ denominator (7).

Прототип функції **assert**:

|  |  |
| --- | --- |
|  | void assert(int expression); |

Заголовний файл: cassert

Функція assert оцінює вираз, яке передається їй як аргумент, через параметр expression. Якщо аргумент-вираз цього макросу у функціональній формі дорівнює нулю (тобто. вираз помилково), повідомлення записується на стандартний пристрій виводу помилок і викликається функція *abort*, яка генерує аварійне завершення програми, робота програми припиняється.

Зміст повідомлення про помилку залежить від конкретної реалізації компілятора, але будь-яке повідомлення повинно складатися з: вираження, яке assert оцінює, ім'я файлу c помилкою і номер рядка, де сталася помилка. Звичайний формат повідомлення про помилку :

ім'я файлу: номер рядка: вираз: повідомлення:

Функція assert не буде доступна, якщо в момент включення <cassert> макрос з ім'ям NDEBUG вже буде визначений. Тобто, при налагодженні програми, ми можемо необмежену кількість разів викликати assert. Після налагодження програми, не буде потрібна assert в викликаних нами функціях. В С визначений макрос NDEBUG, який відключить всі assert функції за нас. Наше завдання – просто включити цю рядок в код програми на початку файлу вихідного коду перед підключенням заголовного файлу:

#define NDEBUG

Параметри: expression

Вираз для оцінки. Якщо логічний вираз в параметрі expression дорівнює 0, функція assert негайно завершує програму.

Більш детально переваги та недоліки assert розглядаються в статті

<https://habr.com/ru/post/141080/>

Якщо ви захочете заблокувати створення об’єктів ваших класів, зробивши в цьому прикладі конструктор копіювання закритим, ви отримаєте помилку компіляції.

Конструктор копіювання може бути проігнорований. Розглянемо приклад.

#include <cassert>

#include <iostream>

class Drob

{

private:

int m\_numerator;

int m\_denominator;

public:

    // Конструктор по умолчанию

    Drob(int numerator=0, int denominator=1) :

        m\_numerator(numerator), m\_denominator(denominator)

    {

        assert(denominator != 0);

    }

        // Конструктор копирования

Drob(const Drob &drob) :

m\_numerator(drob.m\_numerator), m\_denominator(drob.m\_denominator)

{

// Нет необходимости выполнять проверку denominator здесь, так как эта проверка уже осуществляется в конструкторе класса Drob

std::cout << "Copy constructor worked here!\n"; // просто, чтобы показать, что это работает

}

friend std::ostream& operator<<(std::ostream& out, const Drob &d1);

};

std::ostream& operator<<(std::ostream& out, const Drob &d1)

{

out << d1.m\_numerator << "/" << d1.m\_denominator;

return out;

}

int main()

{

Drob sixSeven(Drob(6, 7));

std::cout << sixSeven;

return 0;

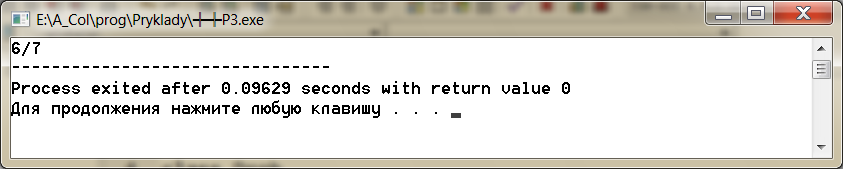
}

Спочатку ініціалізується **анонімний об’єкт** Drob, що викликає конструктор Drob(int, int). Потім цей анонімний об’єкт використовується для ініціалізації об’єкту sixSeven класу Drob. Оскільки анонімний об’єкт є об’єктом класу Drob, як і sixSeven, то тут очікується виклик конструктора копіювання, тобто очікуємий результат:

Copy constructor worked here!

6/7

Реальний результат:



Конструктор копіювання не спрацював тому, що ініціалізація анонімного об’єкта, а потім використання цього об’єкта для прямої ініціалізації вже не анонімного об’єкта виконується в два етапи (перший етап — створення анонімного об’єкта, другий етап — це виклик конструктора копіювання). Але кінцевий результат по суті ідентичний простому виконанню прямої ініціалізації, яка займає лише один крок. З цієї причини в таких випадках компілятору дозволяється відмовитися від виклику конструктора копіювання і просто виконати пряму ініціалізацію. Цей процес називається **елізією**.

**Анонімний об’єкт** — це значення без імені. Оскільки імені немає, то і способу посилатися на цей об’єкт за межами місця, де він створений — теж немає. Отже, анонімні об’єкти мають **область видимості** виразу і вони створюються, обробляються і знищуються в межах одного виразу. Анонімні об’єкти в мові C++ використовуються для передачі або повернення значень без необхідності створювати величезну кількість тимчасових змінних. **Динамічне виділення пам’яті** також виконується через анонімні об’єкти (тому адреса виділеної пам’яті повинна бути присвоєна **покажчику**, інакше ми не мали б можливості посилатися/використовувати її). анонімні об’єкти можна використовувати тільки один раз, так як вони мають область видимості виразу. Якщо вам потрібно посилатися на значення в декількох виразах, то для цього слід використовувати окрему змінну.

**Конструктор і оператор копіювання**

Конструктор копіювання – тип конструктора, що використовується при ініціалізації екземпляру ідентичного (рівного) раніше об’явленому.

Оператор копіювання – тип оператора, призначений для копіювання значень одного екземпляра (правого) у інший (лівий). При цьому обидва екземпляра були раніше створенні.

Аналогічне: створюють два ідентичні екземпляри [побітове копіювання]

Відрізняється: місце виклику і робота з пам’яттю

Конструктор копіювання викликається у рядку, де вказується тип об’єкту, тобто створюється новий об’єкт (виділяється пам’ять).

Оператор копіювання викликається у всіх інших місцях де є операція =. Обидва об’єкти вже створенні раніше і тому він звільняє пам’ять this, а виділяє пам’ять рівну переданому аргументу.

| Тип методу | Прототип | Примітка |
| --- | --- | --- |
| Порожній конструктор | ім’я\_класу(); | Ініціалізує стан наперед визначеними значеннями |
| Конструктор ініціалізації | ім’я\_класу(тип параметр,…) | Тип – будь-який; ініціалізує стан значеннями, що задані у списку аргументів |
| Конструктор копіювання | ім’я\_класу(const ім’я\_класу& параметр); | Ініціалізує стан значенням вказаного в списку аргументів екземпляру даного класу; модифікатор const вказує, що для ініціалізації екземпляру класу можна використовувати константи |

Наведемо приклади оголошення конструкторів:

Class cls1 {

int х, у;       // захищені змінні класу

public:

cls1();         // конструктор по замовчуванню без параметрів

cls1 (int xval=0; int yval=0); /\* конструктор по замовчуванню з параметрами, що мають значення по замовчуванню \*/

cls1(const clsl &pt);    // конструктор копіювання

};

При оголошенні об’єкти ініціюють захищені змінні, тому вони оголошуються з параметрами або без них залежно від виду конструктора. Якщо клас має декілька конструкторів, для кожного об’єкта використовується той, що співпадає з ним за кількістю та типом аргументів.

Якщо в головній програмі оголошено, наприклад:

main() {

cls1 p1; cls1 p2(10,20); cls1 р3(р2);

}……………………………

то у першому випадку буде виконуватися конструктор за замовчуванням без параметрів, у другому — конструктор з параметрами за замовчуванням, у третьому — конструктор копіювання.

**Явний та неявний виклик конструкторів**

class Point

{

static int \_freeID;

const int \_pointID;

double \_x;

double \_y;

public:

Point (double x=0, double y=0);

Point (const Point &);

~Point();

};

**Правило :**  [**Явно забороняйте компілятору генерувати функції, які вам не**](http://cpp.com.ru/meyers_1/ch2.html) **потрібні**

**Приклад класу з двома конструкторами (використання This).**

class Demo

{ int a;

double y;

public Demo( int a ) // конструктор 1

{

this.a = a;

this.y = 0.002;

}

public Demo( double y ) // конструктор 2

{

this.a = 1;

this.y = y;

}

...

}

...

Demo x = new Demo( 300 ); // виклик конструктора 1

Demo y = new Demo( 5.71 ); // виклик конструктора 2

Конструктор – особливий вид метода, що служить для ініціалізації

* **об’єкта** (конструктор екземпляра) або
* **класу** (статичний конструктор).

Конструктор екземпляра ініціалізує дані екземпляра, конструктор класу — дані класу.

**Конструктори екземпляра**

Конструктор викликається автоматично під час створення об’єкта класу за допомогою операції **new**. Ім’я конструктора співпадає з іменем класу.

**Деструктори**

**Деструктор** – це спеціальна функція класу, що виконує всі фіналізуючі дії і викликається неявно компілятором при знищенні екземпляру класу.

**Деструктори** необхідні для автоматичного руйнування об’єктів класу. їх використання має такі особливості:

* ім’я деструктора повинно починатися знаком ~ (тильда), за яким записується ім’я класу;
* для деструктора не вказується тип значення, що повертається (як і для конструкторів);
* клас завжди має тільки один деструктор або не має жодного. У такому випадку компілятор сам створює деструктор за замовчуванням;
* деструктор не може мати параметрів; програма автоматично викликає деструктор, якщо об’єкт класу виходить за межі області дії і повинен бути знищений.

**Головне правило об'єктного програмування -** кожній структурі надаються конструктор і деструктор .

**Друге правило об'єктного програмування -** ніколи не користується конструкторами за замовчуванням, згенерованими системою програмування. Визначивши власні конструктор і деструктор ви повністю контролюєте створення і видалення ваших об'єктів, а не передоручаєте це комусь (віддаєте дітей в дитячий будинок)

**Об'єкт – екземпляр класу.** Об'єкт характеризується ідентичністю, станом і поведінкою.

**Ідентичність** ― це властивість, що відрізняє об'єкт від усіх інших об'єктів. Об'єкт набуває ідентичності при створенні його конструктором і втрачає її при видаленні його деструктором.

**Стан** визначається набором значень атрибутів об'єкту.

**Поведінка** визначається набором методів.

Конструктори і деструктори необхідні для роботи з динамічними даними. Для виділення динамічної пам’яті користуються оператором new у конструкторі, а для її звільнення — оператором delete у деструкторі.

Розглянемо програму, в якій за допомогою конструктора виділяється динамічна пам’ять для рядка символів, ініціюється закрита змінна len, а деструктор звільняє цю пам’ять після її використання.

*Приклад.* Навести програму, в якій треба ввести рядок символів, потім вивести на екран цей рядок і кількість символів у ньому.

**/\*— використання конструкторів и деструкторів при роботе з динамічними даними \*/**

**#include <iostream>**

**#include <string** **.h>**

**#include <conio>**

**using namespace std;**

**class strclass {**

**char \*str;**

**int len;**

**public:**

**strclass(int size)   //—————————- конструктор класу**

**{**

**str=new char[size]; //виділення динамич. пам‘яти для рядка**

**len=size;**

**}**

**~ strclass( )       //———————————–деструктор класу**

**{**

**delete str;         // звільнення динамич. пам‘яти**

**}**

**//——————————— прототипи фукцій введення та виведення**

**int input(void);**

**void output(void);**

**};**

**//—————————————— функція введення рядка**

**int strclass::input(void)**

**{**

**cout<<“Enter row not large “<<len<<” symbols\n”;**

**cin.getline(str, len);**

**return strlen(str);**

**}**

**/——————————————- функція виведення рядка**

**void strclass::output(void) {**

**cout << str << ‘\n’;**

**}**

**int main() {system("color F0");**

**int lenght;**

**strclass my(100);         // Оголошення об‘єкту класу**

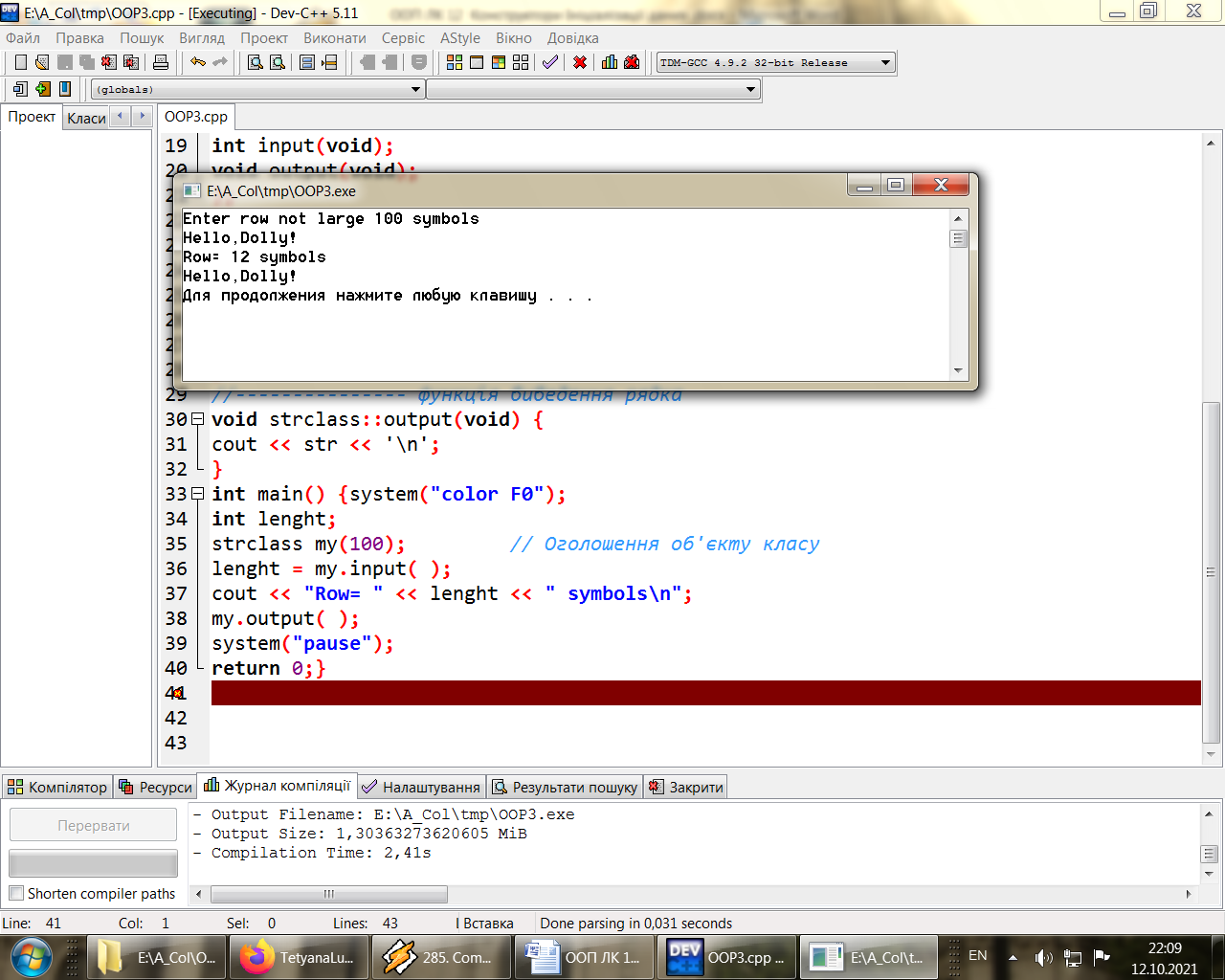
**lenght = my.input( );**

**cout << “Row= ” << lenght << ” symbols\n”;**

**my.output( );**

**system("pause");** **;**

**return 0;}**



**Які функції компілятор створює автоматично?**

Всі ці функції будуть відкритими і вбудованими і виконувати побітове копіювання.

**Зауваження про ініціалізацію екземплярів класу.**

При створенні масиву екземплярів класу виникає необхідність у конструкторі за замовчуванням (без параметрів):

**class Student**

**{**

**private : // закриті члени класу**

**public :**

**Student (double ball\_, double exam\_,**

**char \* name\_ = "NoName");**

**Student (); // конструктор за замовчуванням**

**Student (char \* name\_ = "NoName");**

**~Student ();**

**};**

**int main ()**

**{**

**Student grup\_1 [25]; // виклик констр. за замовч.**

**return 0;**

**}**

А якщо необхідний масив з проініціалізованими полями екземплярів? В такому разі можливо передавати конструкторам аргументи наступним чином:

**int main ()**

**{**

**Student bad [] = {**

**Student (30,25, “Ivan"), // явний виклик**

**Student (32,27, “Oleg"),// явний виклик**

**Student (25,30, "Maria") // явний виклик**

**};**

**return 0;**

**}**

Тут створений масив із трьох елементів, кожний з яких ініціалізується завдяки явному виклику конструктора з параметрами.

***Контрольні запитання*.**

1. Що таке конструктор?
2. Які бувають типи конструкторів?
3. Що таке Конструктор за замовчуванням?
4. Що таке перевантаження конструкторів?
5. Дайте визначення конструктора класу.
6. Для чого призначений конструктор класу?
7. Які види конструкторів ви знаєте?
8. Скільки конструкторів можна оголосити у класі?
9. Як у класі оголосити конструктор з параметрами?
10. Для чого призначений деструктор класу?
11. Скільки деструкторів можна оголосити у класі?
12. Як викликається конструктор і деструктор?
13. Якщо клас має декілька конструкторів, то який з конструкторів викликається при створенні об’єкта цього класу?
14. Чи завжди необхідно описувати в класі деструктор? Якщо ні, то в яких випадках?
15. Особливості створення та ініціалізація об’єктів.

***Для самостійного вивчення***: Розібрати приклад з комплексними числами в главі 4 та задачу в главі 5 [Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997. URL: http://khizha.dp.ua/library/Timothy\_Budd\_-\_Introduction\_to\_OOP\_(ru).pdf]. Поглибити матеріал лекції за наданою літературою.

**Рекомендована література**

1. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL: <http://www.ph4s.ru/bookprogramir_1.html>
2. Страуструп Б. Язык программирования С++. — СПб.; М.: Невский диалект — ЗАО “Изд-во “Бином”, 1999.
3. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..

4. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>

5. Дейтел Х., Дейтел П. Основы программирования на С++. – М.: Бином, 1999. – 1024 с. URL: <http://ijevanlib.ysu.am/wp-content/uploads/2018/03/deytel.pdf>

6. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++ . [2-е изд.] / Буч Г. - СПб.: Невский Диалект, 1998. - 560 с.

7. Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997. URL: <http://khizha.dp.ua/library/Timothy_Budd_-_Introduction_to_OOP_(ru).pdf>

8. *Скотт Мейерс.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. *"ДМК", 2000; 240 с.*

**Скот Маэрс. Эффективное использование С++.**

Глава 1 Приучайтесь к C++

[Правило 1: Относитесь к C++ как к конгломерату языков](http://cpp.com.ru/meyers_1/ch1.html)

  Правило 2: Предпочитайте const, enum и inline использованию #define

  Правило 3: Везде, где только можно используйте const

    Константные функции-члены

  Правило 4: Прежде чем использовать объекты, убедитесь, что они инициализированы

[Глава 2 Конструкторы, деструкторы и операторы присваивания](http://cpp.com.ru/meyers_1/ch2.html)

  Правило 5: Какие функции C++ создает и вызывает молча

  Правило 6: Явно запрещайте компилятору генерировать функции, которые вам не нужны

[Глава 4 Проектирование программ и объявления](http://cpp.com.ru/meyers_1/ch4.html)

  Правило 18: Проектируйте интерфейсы так, что их легко было использовать правильно и трудно – неправильно

  Правило 19: Рассматривайте проектирование класса как проектирование типа

  Правило 20: Предпочитайте передачу по ссылке на const передаче по значению

  Правило 21: Не пытайтесь вернуть ссылку, когда должны вернуть объект

  Правило 22: Объявляйте данные-члены закрытыми

  Правило 23: Предпочитайте функциям-членам функции, не являющиеся ни членами, ни друзьями класса