|  |  |  |  |
| --- | --- | --- | --- |
| Date: | **25-06-2020** | Name: | **Varun G Shetty** |
| Course: | **C ++ programming** | USN: | **4AL17EC093** |
| Topic: | * Constructors,Destructors * Member functions | Semester & Section: | **6th & ‘B’** |
| GitHub Repository: | **Varunshetty4** |  |  |

**Constructors:** Class **constructors**are special member functions of a class. They are executed whenever new objects are created within that class. The constructor's name is identical to that of the class. It has no return type, not even void.

|  |  |
| --- | --- |
| #include <iostream>  #include <string>  using namespace std;  class myClass {  myClass() {  cout <<"Hey";  }  public:  void setName(string x) {  name = x;  }  string getName() {  return name;  }  private:  string name;  };  int main() {  myClass myObj;  myObj.setName("John");  cout << myObj.getName();  return 0;  } | >>Hey john |

**Constructors**can be very useful for setting initial values for certain member variables.  
A default constructor has no parameters. However, when needed, parameters can be added to a constructor. This makes it possible to assign an initial value to an object when it's created, as shown in the following example:

|  |
| --- |
| class myClass {  public:  myClass(string nm) {  setName(nm);  }  void setName(string x) {  name = x;  }  string getName() {  return name;  }  private:  string name;  };  int main() {  myClass ob1("David");  myClass ob2("Amy");  cout << ob1.getName();  }  //Outputs "David" |

We defined a constructor, that takes one parameter and assigns it to the **name**attribute using the **setName()** method. When creating an object, you now need to pass the constructor's parameter, as you would when calling a function.

**Creating a new Class:** It is generally a good practice to define your new classes in separate files. This makes maintaining and reading the code easier. To do this, use the following steps in CodeBlocks:  
Click **File**->**New**->**Class...**  
Give your new class a name, uncheck "Has destructor" and check "Header and implementation file shall be in same folder", then click the "**Create**" button.

The new files act as templates for our new class.

- **MyClass.h** is the **header**file.

- **MyClass.cpp**is the **source**file.

Source & Header: The header file (.h) holds the function declarations (prototypes) and variable declarations. It currently includes a template for our new **MyClass**class, with one default constructor. The implementation of the class and its methods go into the source file (.cpp). Currently it includes just an empty constructor.

|  |  |
| --- | --- |
| **MyClass.h**  #ifndef MYCLASS\_H  #define MYCLASS\_H  class MyClass  {  public:  MyClass();  protected:  private:  };  #endif // MYCLASS\_H | **MyClass.cpp**  #include "MyClass.h"  MyClass::MyClass()  {  //ctor  } |

The **double colon** in the source file (.cpp) is called the **scope resolution operator**, and it's used for the constructor definition. The scope resolution operator is used to define a particular class' member functions, which have already been declared. Remember that we defined the constructor prototype in the **header file**. So, basically, **MyClass::MyClass()**refers to the **MyClass()**member function - or, in this case, constructor - of the **MyClass**class.

To use our classes in our main, we need to include the **header** file. The **header** declares "what" a class (or whatever is being implemented) will do, while the **cpp source** file defines "how" it will perform those features.

**Destructors**: **Destructors**are special functions, as well. They're called when an object is **destroyed**or **deleted**. Objects are destroyed when they go out of scope, or whenever the **delete**expression is applied to a pointer directed at an object of a class. The name of a **destructor**will be exactly the same as the class, only prefixed with a **tilde (~)**. A destructor can't return a value or take any parameters.

After declaring the destructor in the header file, we can write the implementation in the source file MyClass.cpp:. Since destructors can't take parameters, they also can't be overloaded.  
Each class will have just **one**destructor. Defining a destructor is not mandatory; if you don't need one, you don't have to define one.

**Selection Operators:** #ifdef, #ifndef, #endif etc works same as c.

**Pointers to Object:** We can also use a **pointer**to access the object's members.

MyClass obj;

MyClass \*ptr = &obj;

The type of the pointer is **MyClass**,as it points to an object of that type. The **arrow member selection operator (->)** is used instead of dot( . ) to access an object's members with a pointer.

**Constants:** A **constant**is an expression with a fixed value. It cannot be changed while the program is running. Use the **const**keyword to define a constant variable. All constant variables **must**be initialized at the time of their creation.

**Constant Objects**: As with the built-in data types, we can make class objects constant by using the **const**keyword.

const MyClass obj;

All const variables must be initialized when they're created. In the case of classes, this initialization is done via constructors. If a class is not initialized using a parameterized constructor, a public default constructor must be provided - if no public default constructor is provided, a compiler error will occur. Once a const class object has been initialized via the constructor, you cannot modify the object's member variables. This includes both directly making changes to public member variables and calling member functions that set the value of member variables.

Only non-const objects can call non-const functions. A constant object can't call regular functions. Hence, for a constant object to work you need a constant function. To specify a function as a **const**member, the **const**keyword must follow the function prototype, outside of its parameters' closing parenthesis. For **const**member functions that are defined outside of the class definition, the **const**keyword must be used on both the function prototype and definition.

void myPrint() const;

void MyClass::myPrint() const

**Member Initializer:** C++ provides a handy syntax for initializing members of the class called the **member initializer list** (also called a **constructor initializer**).

|  |  |
| --- | --- |
| class MyClass {  public:  MyClass(int a, int b) {  regVar = a;  constVar = b;  }  private:  int regVar;  const int constVar;  }; | class MyClass {  public:  MyClass(int a, int b)  : regVar(a), constVar(b)  {  }  private:  int regVar;  const int constVar;  }; |

This class has two member variables, **regVar**and **constVar**. It also has a constructor that takes two parameters, which are used to initialize the member variables. Running the first code returns an **error**, because one of its member variables is a **constant**, which cannot be assigned a value after declaration.

In cases like this one, a **member initialization list**can be used to assign values to the member variables. the initialization list follows the constructor parameters. The list begins with a **colon**(:), and then lists each variable to be initialized, along with the value for that variable, with a comma to separate them. Use the syntax **variable(value)** to assign values. The initialization list eliminates the need to place explicit assignments in the constructor body. Also, the initialization list does not end with a semicolon.

The member initialization list may be used for regular variables, and must be used for constant variables. Even in cases in which member variables are not constant, it makes good sense to use the member initializer syntax.