|  |  |  |  |
| --- | --- | --- | --- |
| Date: | **24-06-2020** | Name: | **Varun G Shetty** |
| Course: | **C ++ programming** | USN: | **4AL17EC093** |
| Topic: | * Functions * Classes and Objects | Semester & Section: | **6th & ‘B’** |
| GitHub Repository: | **Varunshetty4** |  |  |

**Random Number:** In the C++ standard library, you can access a pseudo random number generator function that's called **rand()**. When used, we are required to include the header **<cstdlib>**. Use the **modulo**(%) operator to generate random numbers within a specific range(cout << **1 + (rand() % 6)** << endl;).

The **srand()** function is used to generate truly random numbers.  
This function allows to specify a **seed** value as its parameter, which is used for the **rand().**

**srand(98);**

A solution to generate truly random numbers, is to use **the current time** as a seed value for the **srand**() function. This example makes use of the **time**() function to get the number of seconds on your system time, and randomly seed the rand() function (we need to include the <**ctime**> header for it).

|  |
| --- |
| #include <iostream>  #include <cstdlib>  #include <ctime>  using namespace std;  int main () {  srand(time(0));  for (int x = 1; x <= 10; x++) {  cout << 1 + (rand() % 6) << endl;  }  } |

**Default Parameters:** When defining a function, you can specify a **default**value for each of the last parameters. If the corresponding argument is missing when you call a function, it uses the **default**value.

**Overloading:** Function **overloading**allows to create multiple functions with the**same name**, so long as they have different parameters.

|  |  |
| --- | --- |
| void printNumber(int a) {  cout << a;  } | void printNumber(float a) {  cout << a;  } |

The first code is effective with **integer**arguments only. Overloading it will make it available for other types, such as **floats**. Now, the same **printNumber()** function name will work for both integers and floats. When overloading functions, the definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You **can not**overload function declarations that differ only by **return**type. The following declaration results in an error.

|  |
| --- |
| int printName(int a) { }  float printName(int b) { }  double printName(int c) { } |

**Recursion**: A **recursive function** in C++ is a function that calls itself.

**Array of Functions:** An array can also be passed to a function as an argument.  
The parameter should be defined as an array using square brackets, when declaring the function.

|  |
| --- |
| void printArray(int arr[], int size) {  for(int x=0; x<size; x++) {  cout <<arr[x]<< endl;  }  }  int main() {  int myArr[3]= {42, 33, 88};  printArray(myArr, 3);  } |

**Function Arguments:** There are two ways to pass arguments to a function as the function is being called.  
**By value:** This method copies the argument's actual value into the function's formal parameter. Here, we can make changes to the parameter within the function without having any effect on the argument. By default, C++ uses call**by value** to pass arguments.

**By reference:** This method copies the argument's reference into the formal parameter. Within the function, the reference is used to access the actual argument used in the call. This means that any change made to the parameter affects the argument.

**Pass-by-reference** copies an argument's address into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. To pass the value by reference, argument **pointers**are passed to the functions just like any other value.

|  |
| --- |
| void myFunc(int \*x) {  \*x = 100;  }  int main() {  int var = 20;  myFunc(&var);  cout << var;  }  // Outputs 100 |

**Objects**: In programming, an object is **self-contained**, with its own **identity**. It is separate from other objects. Each object has its own **attributes**, which describe its current state. Each exhibits its own **behavior**, which demonstrates what they can do.

**Class**: Objects are created using **classes**, which are actually the focal point of OOP. A class can be described as an object's **blueprint**, description, or definition. You can use the same class as a blueprint for creating multiple different objects. Each class has a **name**, and describes **attributes**and **behavior**. In programming, the term **type**is used to refer to a class name: We're creating an object of a particular **type**.

**Method**s: **Method**is another term for a class' behavior. A method is basically a **function**that belongs to a class.

Once we've written the class, we can move on to create objects that are based on that class.  
Each object is called an **instance** of a class. The process of creating objects is called **instantiation**. Begin your class definition with the keyword **class**. Follow the keyword with the class name and the class body, enclosed in a set of curly braces.

class BankAccount {};

Define all **attributes**and **behavior**(or members) in the body of the class, within curly braces.  
You can also define an**access specifier** for members of the class. A member that has been defined using the **public**keyword can be accessed from outside the class, as long as it's anywhere within the scope of the class object. the **dot separator (.)** that is used to access and call the method of the object.

|  |
| --- |
| int main()  {  BankAccount test;  test.sayHi();  } |

**Abstraction:** Data **abstraction**is the concept of providing only essential information to the outside world. It's a process of representing essential features **without including implementation details**.   
**Abstraction**means, that we can have an idea or a concept that is completely separate from any specific instance. It is one of the fundamental building blocks of object oriented programming.

**Encapsulation:** In object orientation, encapsulation means more than simply combining attributes and behavior together within a class; it also means restricting access to the inner workings of that class. The key principle here is that an object only reveals what the other application components require to effectively run the application. All else is kept out of view. This is called **data hiding**. benefits of encapsulation are:

- **Control**the way data is accessed or modified.

- Code is more **flexible**and easy to change with new requirements.

- **Change**one part of code without affecting other part of code.

Access specifiers are used to set access levels to particular members of the class. The three levels of access specifiers are **public**, **protected**, and **private**. A **private**member cannot be accessed, or even viewed, from outside the class; it can be accessed only from within the class. A **public**member function may be used to access the **private**members.