In The Name Of God

C++ Tutorial

Author : Hossein Dehghanipour

References:
- Sololearn.com
- GeeksforGeeks.org
- Youtube - Derek Banas

July - August 2021

Contents

1 Introduction to C++

C++ is a popular cross-platform language that can be used to create high-performance applications, operating systems, browsers, video-games, art applications and so on →It is a general-purpose programming language. For now, remember that the entry point of every C++ program is the main() function, irrespective of what the program does.

2 Installing C++

2.1 Install MinGW

- Go to https://sourceforge.net/projects/mingw/.
- Download the MinGW.
- Install it. While installing it, keep everything as it is. Don't change the depository of it's installation.
- Install/tick these items:
  - **mingw32-base**
  - **mingw32-gcc-g++**
  - **msys-base**
- After clicking on these items, on the left hand side of this page click on **"Installation"** and click on **"Apply Changes"**.
- After everything is downloaded click on **"Close"**.
- Set environment path. The default paths are:
  - **C:\MinGW\bin**
  - **C:\MinGW\msys\1.0\bin**
- Now check the installed C++ version using: **$ g++ --version** in you **CMD**.

2.2 Run a C++ Code in CMD

- **NetBeans** is the suggested IDE for C++. However you can use **VsCode** and use the following commands to compile and run the code.
  - **$ g++ -o program fileName.cpp**
  - **$.\program.exe**

- If you want to run a code with a specific version of C++:
  - **$ g++ -o program -std=c++17 main.cpp** → Using C++ version 17.
  - **$.\program.exe**

3 Basics

3.1 cout

- **cout** is the stream object used to perform output on the standard output device which is usually the display screen. cout is used in combination with the **insertion operator «**. You can add multiple insertion operators after cout.

- In C++, the semicolon is used to terminate a statement. Each statement must end with a semicolon. It indicates the end of one logical expression.

- The **<iostream>** header defines the standard stream objects that input and output data.

- A **namespace** is a declarative region that provides a scope to the identifiers (names of elements) inside it. In our code, the line using namespace **std**; tells the compiler to use the **std (standard) namespace**. The **std** namespace includes features of the **C++ Standard Library**.

- The **cout** object does not insert a line break at the end of the output. One way to print two lines is to use the **endl** manipulator, which will put in a line break.

- The new line character\n can be used as an alternative to endl. Using a single cout statement with as many instances of \n as your program requires will print out multiple lines of text.

- **Identifiers:** An identifier is a name for a variable, function, class, module, or any other user-defined item. An identifier starts with a letter (A-Z or a-z) or an underscore (_), followed by additional letters, underscores, and digits (0 to 9).

- **Case Sensitivity:** The C++ programming language is case-sensitive, so *myVariable* and *myvariable* are two different identifiers.

- **User Input:** To enable the user to input a value, use **cin** in combination with the extraction operator **(»)**. The variable containing the extracted data follows the operator.

- Specifying the data type for a given variable more than once results in a syntax error.

- **Auto keyword:** The auto keyword allows you to automatically deduct the type of the variable being declared. It infers the data type of the variable from its value. Any variable declared with the auto keyword should be initialized at the time of its declaration or there will be an error.

- C++ provides the option of nesting an unlimited number of if/else statements.

- In if/else statements, a single statement can be included without enclosing it into curly braces.

- Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop checks its condition at the bottom of the loop. A do...while loop is similar to a while loop. The one difference is that the do...while loop is guaranteed to execute at least one time.

- The Boolean data type returns just two possible values: true (1) and false (0).

- Several of the basic types, including integers, can be modified using one or more of these type modifiers:

  - **signed:** A signed integer can hold both negative and positive numbers.

  - **unsigned:** An unsigned integer can hold only positive values.

  - **short:** Half of the default size.

  - **long:** Twice the default size.

```cpp
// Sample C++ Code
#include <iostream>
using namespace std;

int main()
{

cout << "Hello world!" << endl;

int num;
cin >> num;

auto x = 4; //integer
    auto y = 3.37; //float
    auto z = "hello"; //string

auto num; //runtime error
num=5;  //runtime error


return 0;

}
```

```cpp
// Your First C++ Program
#include <cstdlib> //C standard library
#include <iostream>
#include <string>
#include <vector>
#include <sstream>

// we can also pass "int argc, char **argv" to our main
int main() {
    std::cout << "Hello World!" <<std::endl;
    int a = 10;
    return 0;
}
```

Output: Hello World!

3.2 argc

```cpp
// Your First C++ Program
#include <cstdlib> //C standard library
#include <iostream>
#include <string>
#include <vector>
#include <sstream>


int main(int argc, char **argv) {
    std::cout << "Hello World!" << std::endl;

    if (argc != 1 ){
        std::cout << "You have entered" << argc << "arguments\n";
    }

    for (int i = 0 ; i < argc ; i++){
        std::cout << argv[i] << "\n";
    }

    return 0;
}
```

Input in Terminal:
» g++ -o program main.cpp
» ./program.exe I Love CPP

Output:
I
Love
CPP

3.3 Primitive Types

```cpp
// Your First C++ Program
#include <cstdlib> //C standard library
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <limits> // newly added

int g_iRandomVariable = 0 ;
// "g" is our standard for Global
// "i" is our standard for Integer

const double PI = 3.14 ;

int main() {
    bool bMarries = true;
    char chAlpha = 'a';
    unsigned short int uInteger = 243;
    long int lIntger = PI * 5 ;
    float fMoney = 34.12385;
    double dbBigFloat = fMoney * 20.6666663;
    //auto whatWillIBe = true;

    std::cout << "Min Int: "<< std::numeric_limits<int>::min()
      << " | Max Int: " <<std::numeric_limits<int>::max()
      <<"\n";


    std::cout << "Min Bool: "<< std::numeric_limits<bool>::min()
      << " | Max Bool: " <<std::numeric_limits<bool>::max()
      <<"\n";


    std::cout << "Size of Int: "<< sizeof(int) << "\n";

```

```
32      printf("%c, %d, %f", chAlpha, lIntger, fMoney);
33
34  }
```

Output:
Min Int: -2147483648 | Max Int: 2147483647
Min Bool: 0 | Max Bool: 1
Size of Int: 4
a, 15, 34.123852

3.4 Basic String Calls

- A string is an ordered sequence of characters, enclosed in double quotation marks.
- It is part of the Standard Library.
- You need to include the **<string>** library to use the string data type. Alternatively, you can use a library that includes the string library. (**#include <string>**)

```
1   // Your First C++ Program
2   #include <cstdlib> //C standard library
3   #include <iostream>
4   #include <string>
5   #include <vector>
6   #include <sstream>
7   #include <limits> // newly added
8
9   int g_iRandomVariable = 0 ;
10  // "g" is our standard for Global
11  // "i" is our standard for Integer
12
13  const double PI = 3.14 ;
14
15  int main() {
16      std::string question ("Enter a number:");
17      std::string num1 = "12", num2 ;
18      std::cout << question;
19      getline(std::cin, num2); //receive user input from keyboard
            "cin"
20      std::cout << "Number 1: " << num1 << " | Number 2: " << num2
            <<"\n" ;
21      int iNum1 = std::stoi(num1); //String to Integer
22      int iNum2 = std::stoi(num2);
```

```
23
24      printf("%d + %d = %d \n", iNum1 , iNum2 , (iNum1 + iNum2));
25      printf("%d - %d = %d \n", iNum1 , iNum2 , (iNum1 - iNum2));
26      printf("%d * %d = %d \n", iNum1 , iNum2 , (iNum1 * iNum2));
27      printf("%d / %d = %f \n", iNum1 , iNum2 , (iNum1 / iNum2));
28      printf("%d Mode %d = %d \n", iNum1 , iNum2 , (iNum1 %
           iNum2));
29  }
```

Output:
Enter a number:25
Number 1: 12 | Number 2: 25
12 + 25 = 37
12 - 25 = -13
12 * 25 = 300
12 / 25 = 0.000000
12 Mode 25 = 12

Another way to code this is defining a **namespace** above the main. This method is not suggested due to some challenges and confusions it brings later on.

```
1   #include <cstdlib>
2   #include <iostream>
3   #include <string>
4   #include <vector>
5   #include <sstream>
6   #include <limits>
7
8   int g_iRandomVariable = 0 ;
9
10
11  const double PI = 3.14 ;
12  using namespace std; //newly added
13  int main() {
14      string question ("Enter a number:");
15      string num1 = "12", num2 ;
16      cout << question;
17      getline(cin , num2);
18      cout << "Number 1: " << num1 << "| Number 2: " << num2
           <<"\n" ;
19      int iNum1 = stoi(num1);
20      int iNum2 = stoi(num2);
21
22      printf("%d + %d = %d \n", iNum1 , iNum2 , (iNum1 + iNum2));
23      printf("%d - %d = %d \n", iNum1 , iNum2 , (iNum1 - iNum2));
24      printf("%d * %d = %d \n", iNum1 , iNum2 , (iNum1 * iNum2));
```

```
25        printf("%d / %d = %f \n", iNum1 , iNum2 , (iNum1 / iNum2));
26        printf("%d Mode %d = %d \n", iNum1 , iNum2 , (iNum1 %
              iNum2));
27    }
```

3.5 Conditional Operators

3.5.1 if, else if, else

```
1   #include <cstdlib>
2   #include <iostream>
3   #include <string>
4   #include <vector>
5   #include <sstream>
6   #include <limits>
7
8
9
10  int main() {
11      std::string sAge = "0";
12      std::cout << "Enter your age: ";
13      getline(std::cin, sAge);
14      int iAge = std::stoi(sAge);
15
16      if( (iAge >= 15) && (iAge <= 18) ){
17          std::cout <<"Suitable";
18      }
19      else if((iAge >= 65) || (iAge <= 5)){
20          std::cout <<"Unsuitable";
21      }
22      else{
23          std::cout <<"No regulations is defined over this age";
24      }
25  }
```

Output:
Enter your age: 36
No regulations is defined over this age

3.5.2 Switch Cases

- In a switch statement, the optional default case can be used to perform a task when none of the cases is determined to be true.

- As we know, if we don't put **break** statements at the end of each **case**, all of the other cases below our target would also be executed.

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

int main () {
   // local variable declaration:
   char grade = 'D';

   switch(grade) {
      case 'A' :
         cout << "Excellent!" << endl;
         break;
      case 'B' :
      case 'C' :
         cout << "Well done" << endl;
         break;
      case 'D' :
         cout << "You passed" << endl;
         break;
      case 'F' :
         cout << "Better try again" << endl;
         break;
      default :
         cout << "Invalid grade" << endl;
   }
   cout << "Your grade is " << grade << endl;

   return 0;
}
```

Output:
You passed
Your grade is D

3.5.3 Ternary Operator

<condition> ? <true-case-code> : <false-case-code>;

```cpp
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
```

```cpp
#include <limits>

int main() {
    char grade = 'E';
    (grade =='A' ? printf("Excellent") :( (grade =='B' || grade
      =='C') ? printf("Well Done") :(grade == 'D' ? printf("You
      Passed") :(grade == 'F' ? printf("Better Try again") :
      printf("Ivalid Grade")))));

    return 0;
}
```

Output:
Ivalid Grade

3.6 Arrays

```cpp
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <limits>

int main() {

    int arrNums[10] = {1};
    int arrNum2[] = {1,2,3};
    int arrNum3[5] = {2,3};

    std::cout << "Arraysize: " <<
       sizeof(arrNum3)/sizeof(*arrNum3) <<std::endl;

    int matrix[3][4][5] = {
    {{0,4,3,2,1},{1,2,3,4,5},{9,7,8,9,1},{0,0,0,0,0}},
  {{1,4,3,2,1},{1,2,3,4,5},{6,7,8,9,1},{0,0,0,0,0}},
  {{2,4,3,2,1},{1,2,3,4,5},{8,7,8,9,1},{0,0,0,0,0}},
    };

    std::cout << "Arraysize: " <<
       sizeof(matrix)/sizeof(*matrix)<<std::endl;

    std::cout << "Arraysize: " << matrix[1][2] <<std::endl;

}
```

Output:
Arraysize: 5
Arraysize: 3
Arraysize: 0x72fd28

3.7 Vectors

3.7.1 Initializing

```cpp
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <limits>

int main() {
std::vector<int> numbers(5); //allocated 5 int places for this
   vector
numbers[0] = 10;
numbers[1] = 20;
numbers.push_back(30); // pushes the value to the latest index

std::cout << "Last Item: " << numbers[numbers.size() - 1];

}
```

Output:
Last Item: 30

3.7.2 Push, Pop, Empty

void push_back (const valueType);
void pop_back();

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(void) {
   vector<int> v;
```

```
8
9       /* Insert 5 elements */
10      for (int i = 0; i < 5; ++i){
11           v.push_back(i + 1);
12      }
13
14       for (int i = 0; i < v.size(); ++i){
15           cout << v[i] << endl;
16           if (v.empty() == false){
17                v.pop_back(); //returns Void
18           }
19       }
20
21      return 0;
22  }
```

Output:
1
2
3

3.7.3 Size and Capacity

```
1   #include <iostream>
2   #include <vector>
3
4   using namespace std;
5
6   int main(void) {
7      vector<int> v;
8
9      for (int i = 0; i < 5; ++i)
10        v.push_back(i + 1);
11
12     cout << "Number of elements in vector = " << v.size() << endl;
13     cout << "Capacity of vector           = " << v.capacity() <<
          endl;
14
15     return 0;
16  }
```

Output:
Number of elements in vector = 5
Capacity of vector = 8

3.7.4 Begin, End, Erase

iterator erase (const_iterator position);
iterator erase (const_iterator first, const_iterator last); →Removes items in between the ranges
const_iterator end() const noexcept;
iterator end() noexcept;
iterator begin() noexcept;
const_iterator begin() const noexcept;

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(void) {
    vector<int> v = {1, 2, 3, 4, 5};

    cout << "Original vector" << endl;
    for (auto it = v.begin(); it != v.end(); ++it)
        cout << *it << endl;

    /* Remove first element */
    v.erase(v.begin());

    cout << "Modified vector" << endl;
    for (auto it = v.begin(); it != v.end(); ++it)
        cout << *it << endl;

    return 0;
}
```

Output:
Original vector
1
2
3
4
5
Modified vector
2
3
4
5

3.7.5 Clear

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(void) {
    auto ilist = {1, 2, 3, 4, 5};
    vector<int> v(ilist);

    cout << "Initial size of vector     = " << v.size() << endl;
    /* destroy vector */
    v.clear();
    cout << "Size of vector after clear = " << v.size() << endl;

    return 0;
}
```

Output:
Initial size of vector = 5
Size of vector after clear = 0

3.8 Foreach Loop

```cpp
// C++ program to demonstrate use of foreach
#include <iostream>
using namespace std;

int main()
{
```

```
7     int arr[] = { 10, 20, 30, 40 };
8
9     // Printing elements of an array using
10    // foreach loop
11    for (int x : arr)
12      cout << x << endl;
13  }
14  void printArr(std::vector<int> vec){
15      for (auto item: vec){
16          std::cout<< item <<std::endl;
17      }
18  }
```

Output:
10
20
30
40

4 Pointers and Functions

As we already know, in C/C++ languages, the functions must be declared before calling. Despite Java and Python, the functions must be declared above the usage but can be defined below the calling.

4.1 Function Basics

- Using functions can have many advantages, including the following:
  - You can reuse the code within a function.
  - You can easily test individual functions.
  - If it's necessary to make any code modifications, you can make modifications within a single function, without altering the program structure.
  - You can use the same function for different inputs.
- Every valid C++ program has at least one function - the main() function.
- You must declare a function's prototype before calling it but defining it can come after using it. (A function declaration, or function prototype, tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.)
- Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.
- The prototype in both declaration and definition should match.
- 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>**.
- 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()** function's algorithm. **Changing the seed value changes the return of rand(). However, the same argument will result in the same output.**
- 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). **time(0)** will return the current second count, prompting the srand() function to set a different seed for the rand() function each time the program runs.

4.1

```

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

int main() {
  cout << rand();
}
```

4.1

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

int main () {
    srand(98);

    for (int x = 1; x <= 10; x++) {
        cout << 1 + (rand() % 6) << endl;
    }
}
```

```

#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;
    }
}
```

```
/*Fill in the blanks to print to the screen a truly random
   number from 1 through 15, generated by the rand() function.*/

#include <iostream>
```

```
#include <cstdlib>
#include <ctime>
using namespace std;
int main () {
    srand(time(0));
    cout << 1 + (rand() % 15) << endl;

}
```

4.2 Default Values for 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.

- To do this, use the assignment operator to assign values to the arguments in the function definition, as shown in this example.

```

int sum(int a, int b=42) {
  int result = a + b;
  return (result);
}
```

```



#include <iostream>
using namespace std;

int volume(int l=1, int w=1, int h=1) {
    return l*w*h;
}

int main() {
    cout << volume() << endl;
    cout << volume(5) << endl;
    cout << volume(2, 3) << endl;
    cout << volume(3, 7, 6) << endl;
}
```

Output:
1
5
6
126

4.3 Overloading

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

- 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.

- As you can see, the function call is based on the argument provided. An integer argument will call the function implementation that takes an integer parameter. A float argument will call the implementation taking a float parameter.

- You can not overload function declarations that differ only by return type.

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

void printNumber(int x) {
    cout << "Prints an integer: " << x << endl;
}
void printNumber(float x) {
    cout << "Prints a float: " << x << endl;
}

int main() {
    int a = 16;
    float b = 54.541;
    printNumber(a);
    printNumber(b);
}
```

Output:
Prints an integer: 16
Prints a float: 54.541

4.4 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. Because a copy of the argument is passed to the function, the original argument is not modified by the function.

- **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.opies 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.

- By default, C++ uses call by value to pass arguments.

- In general, passing by value is faster and more effective. Pass by reference when your function needs to modify the argument, or when you need to pass a data type, that uses a lot of memory and is expensive to copy.

```
1  // Pass by Value Example
2  #include <iostream>
3  using namespace std;
4
5  void myFunc(int x) {
6      x = 100;
7  }
8
9  int main() {
10     int var = 20;
11     myFunc(var);
12     cout << var;
13 }
```

Output:
20

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

void myFunc(int *x) {
    *x = 100;
}

int main() {
    int var = 20;
    myFunc(&var);
    cout << var;
}
```

Output:
100

As you can see, we passed the variable directly to the function using the address-of operator **&**. The function declaration says that the function takes a pointer as its parameter (defined using the * operator).

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <numeric>
#include <string>


std::string isEven(int);
bool isOdd(int);

int main(void) {
   std::vector<int> v(10);

   std::iota(std::begin(v), std::end(v), 0);
   // starts from 0 and goes up untill the vector is full

   for (int i = 0 ; i < v.size() ; i++){
       std::cout << v[i] << " | is Even?: " << isEven(v[i]) << "
          | Is Odd?:" << isOdd(v[i]) << std::endl;
   }



   return 0;

}
```

```
27  std::string isEven(int number){
28      if ( (number % 2) == 0){
29          return "true";
30      }
31      return "false";
32  }
33
34  bool isOdd(int number){
35
36      if ( (number % 2) == 0){
37          return true;
38      }
39      return false;
40  }
```

Output:
0 | is Even?: true | Is Odd?:1
1 | is Even?: false | Is Odd?:0
2 | is Even?: true | Is Odd?:1
3 | is Even?: false | Is Odd?:0
4 | is Even?: true | Is Odd?:1
5 | is Even?: false | Is Odd?:0
6 | is Even?: true | Is Odd?:1
7 | is Even?: false | Is Odd?:0
8 | is Even?: true | Is Odd?:1
9 | is Even?: false | Is Odd?:0

```
1   \\ Palindrome  Checker
2   #include <iostream>
3   #include <cstring>
4   #include<string>
5   using namespace std;
6
7   bool isPalindrome(int x) {
8       //complete the function
9       string str = to_string(x);
10      int n = str.length();
11      char char_array[n + 1];
12      strcpy(char_array, str.c_str());
13      int last_index = n-1 ;
14      for(int index = 0; index < n/2 ; index++ , last_index--){
15          if(char_array[index] != char_array[last_index]){
16              return false;
17          }
18      }
19
```

```
20      return true;
21  }
22
23  int main() {
24      int n;
25      cin >>n;
26
27      if(isPalindrome(n)) {
28          cout <<n<<" is a palindrome";
29      }
30      else {
31          cout << n<<" is NOT a palindrome";
32      }
33      return 0;
34  }
```

Output:
123 is NOT a palindrome
1235321 is a palindrome
123321 is a palindrome

4.5 Pointers

- There are two operators for pointers:
  - **Address-of** operator (**&**): returns the memory address of its operand.
  - **Contents-of** (or **dereference**) operator (*): returns the value of the variable located at the address specified by its operand.

```
1   #include <iostream>
2   using namespace std;
3   int main()
4   {
5       int var = 50;
6       int  *p;
7       p = &var;
8
9       cout << var << endl;
10      // Outputs 50 (the value of var)
11
12      cout << p << endl;
13      // Outputs 0x29fee8 (var's memory location)
```

```
14
15      cout << *p << endl;
16      /* Outputs 50 (the value of the variable
17       stored in the pointer p) */
18
19      return 0;
20  }
```

Output:
50
0x7ffdd4b03174
50

```
1   #include <iostream>
2   #include <vector>
3   #include <cstdlib>
4   #include <numeric>
5   #include <string>
6
7
8   std::string isEven(int);
9   void pop(std::vector<int>);
10  void printArr(std::vector<int>);
11  void referencedPop (std::vector<int> &vect);
12
13  int main(void) {
14     std::vector<int> v(10);
15
16      for (int i = 0 ; i < 10 ; i++){
17          v[i] = i ;
18      }
19
20      std::cout << "Size: " << v.size() <<std::endl ;
21      std::cout << "==========Before Normal Pop==========="
          <<std::endl ;
22      printArr(v);
23
24      pop(v);
25
26      std::cout << "==========After Normal Pop==========="
          <<std::endl ;
27      printArr(v);
28      std::cout << "Size: " << v.size() <<std::endl;
29
30
31      referencedPop(v);
```

```
32
33      std::cout << "==========After Referenced Pop==========="
            <<std::endl ;
34      printArr(v);
35      std::cout << "Size: " << v.size() <<std::endl;
36
37
38     return 0;
39
40  }
41
42  std::string isEven(int number){
43      if ( (number % 2) == 0){
44          return "true";
45      }
46      return "false";
47  }
48
49  void pop(std::vector<int> sampleVector){
50      sampleVector.pop_back();
51  }
52
53  void referencedPop (std::vector<int> &vect){
54      vect.pop_back();
55  }
56
57  void printArr(std::vector<int> vec){
58      for (auto item: vec){
59          std::cout<< item <<std::endl;
60      }
61  }
```

```
Output:
Size: 10
==========Before Normal Pop==========
0
1
2
3
4
5
6
7
8
9
==========After Normal Pop==========
0
1
2
3
4
5
6
7
8
9
Size: 10
==========After Referenced Pop==========
0
1
2
3
4
5
6
7
8
Size: 9
```

The pointers are exactly the same as C languages. **vector<int>** is **non-array**, **non-reference**, and **non-pointer** - it is being passed by **value**, and hence it will call copy-constructor. So, you must use **vector<int>&** (preferably with const , if function isn't modifying it) to pass it as a reference.

5 String and Math

5.1 String

```cpp
// ----- STRING TUTORIAL -----

#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <numeric>
#include <sstream>
    #include <algorithm> // for transform method


int main() {

    // A C character string is an array of characters
    // with a null character at the end \0
    char cString[] = {'A', ' ', 'S', 't', 'r', 'i', 'n', 'g',
       '\0'};
    std::cout << cString << "\n";

    // Get array size (null is included)
    std::cout << "Array Size " << sizeof(cString) << "\n";

    // C strings are troublesome because if you forget \0, or
    // add to much information it can lead your program to crash,
    // or for your system to crash

    // You can create a vector of strings
    std::vector<std::string> strVec(10);

    // C++ std::string can grow in size and is much safer
    std::string str("I'm a string");
    strVec[0] = str;

    // You can access characters with an index
    std::cout << str[0] << "\n";

    // You can also use at()
    std::cout << str.at(0) << "\n";

    // Front returns first char and back returns last
    std::cout << str.front() << " " << str.back() << "\n";

    // Get the string length
    std::cout << "Length : " << str.length() << "\n";
```

```
44
45      // You can copy a string to another
46      std::string str2(str);
47      strVec[1] = str2;
48
49      // You can copy after the 1st 4 characters
50      std::string str3(str, 4);
51      strVec[2] = str3;
52
53      // Repeat a value to make a string
54      std::string str4(5, 'x');
55      strVec[3] = str4;
56
57      // Combine strings with append or +
58      strVec[4] = str.append(" and your not");
59      str += " and your not";
60
61      // Append part of a string
62      str.append(str, 34, 37);
63      strVec[5] = str;
64
65      // Erase characters from a string from an index to another
66      // or the last
67      str.erase(13, str.length() - 1);
68      strVec[6] = str;
69
70      for(auto y: strVec)
71          std::cout << y << "\n";
72
73      // find() returns index where pattern is found
74      // or npos
75      if(str.find("string") != std::string::npos)
76          std::cout << "1st not " << str.find("string") << "\n";
77
78      // substr(x, y) returns a substring starting at
79      // index x with a length of y
80      std::cout << "Substr " << str.substr(6,6) << "\n";
81
82      // Reverse a string by passing the beginning and end
83      // of a string
84      reverse (str.begin(), str.end());
85      std::cout << "Reverse " << str << "\n";
86
87      // Case conversion
88      transform(str2.begin(), str2.end(), str2.begin(), ::toupper);
89      std::cout << "Upper " << str2 << "\n";
90      transform(str2.begin(), str2.end(), str2.begin(), ::tolower);
91      std::cout << "Lower " << str2 << "\n";
```

```
92
93      // You can get the ascii code for a char by saving
94      // the char as an int or with (int)
95      // a - z : 97 - 122
96      // A - Z : 65 - 90
97      char aChar = 'Z';
98      int aInt = aChar;
99      std::cout << "A Code " << (int)'a' << "\n";
100
101     // Convert int to string
102     std::string strNum = std::to_string(1+2);
103     std::cout << "String " << strNum << "\n";
104
105     // ----- PROBLEM : SECRET STRING -----
106     // Receive an uppercase string and hide its meaning
107     // by turning it into ascii codes
108     // Translate it back to the original letters
109
110     std::string normalStr, secretStr = "";
111     std::cout << "Enter your string in uppercase : ";
112     std::cin >> normalStr;
113
114     // Cycle through each character converting
115     // them into ascii codes which are stored in
116     // a string
117     for(char c: normalStr)
118         secretStr += std::to_string((int)c);
119         // secretStr += std::to_string((int)c - 23);
120
121     std::cout << "Secret : " << secretStr << "\n";
122
123     normalStr = "";
124
125     // Cycle through numbers in string 2 at a time
126     for(int i = 0; i < secretStr.length(); i += 2){
127
128         // Get the 2 digit ascii code
129         std::string sCharCode = "";
130         sCharCode += secretStr[i];
131         sCharCode += secretStr[i+1];
132
133         // Convert the string into int
134         int nCharCode = std::stoi(sCharCode);
135
136         // Convert the int into a char
137         char chCharCode = nCharCode;
138         // char chCharCode = nCharCode + 23;
139
```

```
140             // Store the char in normalStr
141             normalStr += chCharCode;
142         }
143 
144         std::cout << "Original : " << normalStr << "\n";
145 
146         // ----- END OF PROBLEM : SECRET STRING -----
147 
148         // ----- BONUS PROBLEM -----
149         // Allow the user to enter upper and lowercase
150         // letters by subtracting and adding 1 value
151         // ----- END OF BONUS PROBLEM -----
152 
153         return 0;
154 
155     }
156 
157     // ----- END OF STRING TUTORIAL ----
```

Output:

```
A String
Array Size 9
I
I
I g
Length : 12
I'm a string
I'm a string
a string
xxxxx
I'm a string and your not
I'm a string and your not and your not not
I'm a string

1st not 6
Substr string
Reverse gnirts a m'I
Upper I'M A STRING
Lower i'm a string
A Code 97
String 3
Enter your string in uppercase : ALPHABETS IN UPPER
Secret : 657680726566698483
Original : ALPHABETS
```

5.2 Math

```cpp
// ----- MATH FUNCTIONS -----
// C++ has numerous math functions
// http://en.cppreference.com/w/cpp/numeric/math

// Needed for math functions
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <numeric>
#include <sstream>
int main() {

    std::cout << "abs(-10) = " << std::abs(-10) << "\n";

    std::cout << "max(5,4) = " << std::max(5,4) << "\n";

    std::cout << "min(5,4) = " << std::min(5,4) << "\n";

    std::cout << "fmax(5.3,4.3) = " << std::fmax(5.3,4.3) <<
       "\n";

    std::cout << "fmin(5.3,4.3) = " << std::fmin(5.3,4.3) <<
       "\n";

    // e ^ x
    std::cout << "exp(1) = " << std::exp(1) << "\n";

    // 2 ^ x
    std::cout << "exp2(1) = " << std::exp2(1) << "\n";

    // e * e * e ~= 20 so log(20.079) ~= 3
    std::cout << "log(20.079) = " << std::log(20.079) << "\n";

    // 10 * 10 * 10 = 1000, so log10(1000) = 3
    std::cout << "log10(1000) = " << std::log10(1000)
              << "\n";

    // 2 * 2 * 2 = 8
    std::cout << "log2(8) = " << std::log2(8)
              << "\n";

    // 2 ^ 3
    std::cout << "pow(2,3) = " << std::pow(2,3)
              << "\n";
```

```
45
46      // Returns what times itself equals the provided value
47      std::cout << "sqrt(100) = " << std::sqrt(100)
48              << "\n";
49
50      // What cubed equals the provided
51      std::cout << "cbrt(1000) = " << std::cbrt(1000)
52              << "\n";
53
54      // Hypotenuse : SQRT(A^2 + B^2)
55      std::cout << "hypot(2,3) = " << std::hypot(2,3)
56              << "\n";
57
58      std::cout << "ceil(10.45) = " << std::ceil(10.45)
59              << "\n";
60
61      std::cout << "floor(10.45) = " << std::floor(10.45)
62              << "\n";
63
64      std::cout << "round(10.45) = " << std::round(10.45)
65              << "\n";
66
67      // Also sin, cos, tan, asin, acos, atan, atan2,
68      // sinh, cosh, tanh, asinh, acosh, atanh
69
70      return 0;
71  }
72
73  // ----- END OF MATH FUNCTIONS -----
```

Output:
```
abs(-10) = 10
max(5,4) = 5
min(5,4) = 4
fmax(5.3,4.3) = 5.3
fmin(5.3,4.3) = 4.3
exp(1) = 2.71828
exp2(1) = 2
log(20.079) = 2.99967
log10(1000) = 3
log2(8) = 3
pow(2,3) = 8
sqrt(100) = 10
cbrt(1000) = 10
hypot(2,3) = 3.60555
ceil(10.45) = 11
floor(10.45) = 10
round(10.45) = 10
```

6 Classes and Objects

- Objects also have characteristics that are used to describe them. For example, a car can be red or blue, a mug can be full or empty, and so on. These characteristics are also called**attributes**. An attribute describes the current **state** of an object.

- An object's state is independent of its type; a cup might be full of water, another might be empty.

- So, the following three dimensions describe any object in object oriented programming: **identity (Objects Name), attributes (Entities inside an Object), behavior (Methods inside the Object)**.

- Objects are created using classes, which are actually the focal point of OOP.

- The class describes what the object will be, but is separate from the object itself.

- In other words, 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. For example, in preparation to creating a new building, the architect creates a blueprint, which is used as a basis for actually building the structure. That same blueprint can be used to create multiple buildings.

- Programming works in the same fashion. We first define a class, which becomes the blueprint for creating 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.

- **Attributes** are also referred to as **properties** or **data**.

- A **method** is a function declared inside a 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. **A class definition must be followed by a semicolon.**

- 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. You can also designate a class' members as **private** or **protected**.

- We must declare a class before using it, as we do with functions.

Example:
name: BankAccount attributes: accountNumber, balance, dateOpened behavior: open(), close(), deposit()

```
class BankAccount {

};
```

6.1 Create a Class

```

#include <iostream>
using namespace std;

class BankAccount {
    public:
        void sayHi() {
            cout << "Hi" << endl;
        }
};

int main()
{
    BankAccount test;
    test.sayHi();
}
```

Output:
Hi

6.2 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. A good real-world example is a book: When you hear the term book, you don't know the exact specifics, i.e.: the page count, the color, the size, but you understand the idea of a book - the abstraction of the book. The concept of abstraction is that we focus on essential qualities, rather than the specific characteristics of one particular example. 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. For example, when you use cout, you're actually using the cout object of the class ostream. This streams data to result in standard output. In this example, there is no need to understand how cout will display the text on the user's screen. The only thing you need to know to be able to use it is the public interface. **cout « "Hello!" « endl;**. Abstraction acts as a foundation for the other object orientation fundamentals, such as **inheritance** and **polymorphism**.

6.3 Encapsulation

Part of the meaning of the word encapsulation is the idea of **"surrounding"** an entity, not just to keep what's inside together, but also to **protect** it. 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.
This is also known as "black boxing", which refers to closing the inner working zones of the object, except of the pieces that we want to make public.
This allows us to change attributes and implementation of methods without altering the overall program. For example, we can come back later and change the data type of the balance attribute.

In summary the 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 public member is accessible from outside the class, and anywhere within the scope of the class object.
The name attribute is public; it can be accessed and modified from outside the code.
Access modifiers only need to be declared once; multiple members can follow a single access modifier.
Notice the **colon (:)** that follows the public keyword.
If no access specifier is defined, all members of a class are set to **private** by default. For private attributes, we define **getters** and **setters**

6.4 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.
The constructor is called when an object of the class is being declared.

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

class myClass {
    public:
        myClass() {
            cout <<"Hey";
        }
        void setName(string x) {
            name = x;
        }
        string getName() {
            return name;
        }
    private:
        string name;
};

int main() {
    myClass myObj;

    return 0;
}
```

Output:
Hey

We can also pass some parameters to a constructor.

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

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();
}
```

Output:
David

6.5 The "new" Keyword

This piece of article is copied from GeeksforGeeks.org.
When you create a new object, memory is allocated using operator new function and then the constructor is invoked to initialize the memory. Here, The new operator does both the allocation and the initialization, where as the operator new only does the allocation. Let us see how these both work individually.

6.5.1 New Operator

The new operator is an **operator** which denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable. When you create an object of class using new keyword(normal new).

- The memory for the object is allocated using operator new from heap.

- The constructor of the class is invoked to properly initialize this memory.

```
// CPP program to illustrate
// use of new keyword
#include<iostream>
using namespace std;
class car
{
  string name;
  int num;

```

```
10      public:
11        car(string a, int n)
12        {
13          cout << "Constructor called" << endl;
14          this ->name = a;
15          this ->num = n;
16        }
17
18        void enter()
19        {
20          cin>>name;
21          cin>>num;
22        }
23
24        void display()
25        {
26          cout << "Name: " << name << endl;
27          cout << "Num: " << num << endl;
28        }
29    };
30
31    int main()
32    {
33      // Using new keyword
34      car *p = new car("Honda", 2017);
35      p->display();
36    }
```

Output:
Constructor called
Name: Honda
Num: 2017

6.5.2 Operator New

Operator new is a **function** that allocates raw memory and conceptually a bit similar to **malloc()**.

- It is the mechanism of overriding the default heap allocation logic.
- It doesn't initializes the memory i.e constructor is not called. However, after our overloaded new returns, the compiler then automatically calls the constructor also as applicable.
- It's also possible to overload operator new either globally, or for a specific class

```cpp
// CPP program to illustrate
// use of operator new
#include<iostream>
#include<stdlib.h>

using namespace std;

class car
{
  string name;
  int num;

  public:

    car(string a, int n)
    {
      cout << "Constructor called" << endl;
      this->name = a;
      this->num = n;
    }

    void display()
    {
      cout << "Name: " << name << endl;
      cout << "Num: " << num << endl;
    }

    void *operator new(size_t size)
    {
      cout << "new operator overloaded" << endl;
      void *p = malloc(size);
      return p;
    }

    void operator delete(void *ptr)
    {
      cout << "delete operator overloaded" << endl;
      free(ptr);
    }
};

int main()
{
  car *p = new car("HYUNDAI", 2012);
  p->display();
  delete p;
}
```

Output:
new operator overloaded
Constructor called
Name:HYUNDAI
Num:2012
delete operator overloaded

Attention

- Operator vs function: new is an operator as well as a keyword whereas operator new is only a function.
- New calls "Operator new": "new operator" calls "operator new()" , like the way + operator calls operator +().
- "Operator new" can be Overloaded: Operator new can be overloaded just like functions allowing us to do customized tasks.
- Memory allocation: 'new expression' call 'operator new' to allocate raw memory, then call constructor.

6.6 Scope Resolution Operator

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.

```
#include "MyClass.h"

MyClass::MyClass()
{
    //ctor
}
```

To use our classes in our main, we need to include the header file. For example, to use our newly created MyClass in main:

```
#include <iostream>
#include "MyClass.h"
```

```
using namespace std;

int main() {
  MyClass obj;
}
```

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.

6.7 Destructor

Remember constructors? They’re special member functions that are automatically called when an object is created. **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**. Destructors can be very useful for releasing resources before coming out of the program. This can include closing files, releasing memory, and so on.

```

class MyClass {
  public:
    ~MyClass() {
     // some code
    }
};
```

For example, let’s declare a destructor for our MyClass class, in its header file MyClass.h:

```

class MyClass
{
  public:
   MyClass();
   ~MyClass();
};
```

After declaring the destructor in the header file, we can write the implementation in the source file MyClass.cpp:

```

#include "MyClass.h"
#include <iostream>
using namespace std;

MyClass::MyClass()
```

```
7   {
8     cout<<"Constructor"<<endl;
9   }
10
11  MyClass::~MyClass()
12  {
13    cout<<"Destructor"<<endl;
14 }
```

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.

6.8 Selection Operator

```
1   #ifndef MYCLASS_H
2   #define MYCLASS_H
3
4   class MyClass
5   {
6     public:
7     MyClass();
8     protected:
9     private:
10  };
11
12  #endif // MYCLASS_H
```

ifndef stands for "if not defined". The first pair of statements tells the program to define the **MyClass** header file if it has not been defined already. **endif** ends the condition. This prevents a header file from being included more than once within one file.

```
1   //MyClass.h
2   class MyClass
3   {
4     public:
5      MyClass();
6      void myPrint();
7   };
8
9   //MyClass.cpp
10  #include "MyClass.h"
11  #include <iostream>
12  using namespace std;
13
14  MyClass::MyClass() {
```

```
}

void MyClass::myPrint() {
  cout <<"Hello"<<endl;
}

//Main.cpp
int main() {
    MyClass obj;
    obj.myPrint();
}
```

It's necessary to specify its return type in both the declaration and the definition.
We can also use a pointer to access the object's members.
The following pointer points to the obj object:

```
MyClass obj;
MyClass *ptr = &obj;
```

The arrow member selection operator (->) is used to access an object's members with a pointer.

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

class MyClass
{
    public:
        MyClass();
        void myPrint();
};

MyClass::MyClass() {
}
void MyClass::myPrint() {
    cout <<"Hello"<<endl;
}

int main() {
    MyClass obj;
    MyClass *ptr = &obj;
    ptr->myPrint();
}
```

- When working with an **object**, use the **dot(.)** member selection operator .

- When working with a **pointer** to the object, use the **arrow (->)** member selection operator.

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. **const MyClass obj;**. When you've used const to declare an object, you can't change its data members during the object's lifetime.

6.9 Constructors and member initializer lists

This section is copied from GeeksforGeeks.org.
Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon. Following is an example that uses the initializer list to initialize x and y of Point class.

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

class Point {
private:
  int x;
  int y;
public:
  Point(int i = 0, int j = 0):x(i), y(j) {}
  /* The above use of Initializer list is optional as the
    constructor can also be written as:
    Point(int i = 0, int j = 0) {
      x = i;
      y = j;
    }
  */

  int getX() const {return x;}
  int getY() const {return y;}
};

int main() {
Point t1(10, 15);
cout<<"x = "<<t1.getX()<<", ";
cout<<"y = "<<t1.getY();
return 0;
}

/* OUTPUT:
x = 10, y = 15
*/
```

The above code is just an example for syntax of the Initializer list. In the above code, x and y can also be easily initialed inside the constructor. But there are situations where initialization of data members

inside constructor doesn't work and Initializer List must be used. Following are such cases:

6.9.1 For initialization of non-static const data members

const data members must be initialized using Initializer List. In the following example, "t" is a const data member of Test class and is initialized using Initializer List. Reason for initializing the const data member in initializer list is because no memory is allocated separately for const data member, it is folded in the symbol table due to which we need to initialize it in the initializer list.
Also, it is a Parameterised constructor and we don't need to call the assignment operator which means we are avoiding one extra operation.

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

class Test {
  const int t;
public:
  Test(int t):t(t) {} //Initializer list must be used
  int getT() { return t; }
};

int main() {
  Test t1(10);
  cout<<t1.getT();
  return 0;
}

/* OUTPUT:
10
*/
```

6.9.2 For initialization of reference members

Reference members must be initialized using Initializer List. In the following example, "t" is a reference member of Test class and is initialized using Initializer List.

```
// Initialization of reference data members
#include<iostream>
using namespace std;

class Test {
  int &t;
public:
  Test(int &t):t(t) {} //Initializer list must be used
  int getT() { return t; }
```

```
};

int main() {
  int x = 20;
  Test t1(x);
  cout<<t1.getT()<<endl;
  x = 30;
  cout<<t1.getT()<<endl;
  return 0;
}
/* OUTPUT:
  20
  30
*/
```

6.9.3 For initialization of member objects which do not have default constructor

In the following example, an object “a” of class “A” is data member of class “B”, and “A” doesn’t have default constructor. Initializer List must be used to initialize “a”.

```

#include <iostream>
using namespace std;

class A {
  int i;
public:
  A(int );
};

A::A(int arg) {
  i = arg;
  cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B contains object of A
class B {
  A a;
public:
  B(int );
};

B::B(int x):a(x) { //Initializer list must be used
  cout << "B's Constructor called";
}

```

```
int main() {
  B obj(10);
  return 0;
}
/* OUTPUT:
  A's Constructor called: Value of i: 10
  B's Constructor called
*/
```

If class A had both default and parameterized constructors, then Initializer List is not must if we want to initialize “a” using default constructor, but it is must to initialize “a” using parameterized constructor.

6.9.4 For initialization of base class members

Like point 3 the parameterized constructor of the base class can only be called using Initializer List.

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

class A {
  int i;
public:
  A(int );
};

A::A(int arg) {
  i = arg;
  cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B is derived from A
class B: A {
public:
  B(int );
};

B::B(int x):A(x) { //Initializer list must be used
  cout << "B's Constructor called";
}

int main() {
  B obj(10);
  return 0;
}
```

6.9.5 When constructor's parameter name is same as data member

If constructor's parameter name is same as data member name then the data member must be initialized either using this pointer or Initializer List. In the following example, both member name and parameter name for A() is "i".

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

class A {
  int i;
public:
  A(int );
  int getI() const { return i; }
};

A::A(int i):i(i) { } // Either Initializer list or this pointer
   must be used
/* The above constructor can also be written as
A::A(int i) {
  this->i = i;
}
*/

int main() {
  A a(10);
  cout<<a.getI();
  return 0;
}
/* OUTPUT:
  10
*/
```

6.9.6 For Performance reasons

It is better to initialize all class variables in Initializer List instead of assigning values inside body. Consider the following example:

```
// Without Initializer List
class MyClass {
  Type variable;
public:
  MyClass(Type a) { // Assume that Type is an already
          // declared class and it has appropriate
          // constructors and operators
    variable = a;
```

```
    }
};
```

Here compiler follows following steps to create an object of type MyClass:

1. Type's constructor is called first for "a".

2. The assignment operator of "Type" is called inside body of MyClass() constructor to assign. **variable = a;**

3. And then finally destructor of "Type" is called for "a" since it goes out of scope.

Now consider the same code with MyClass() constructor with Initializer List

```
// With Initializer List
class MyClass {
  Type variable;
public:
  MyClass(Type a):variable(a) { // Assume that Type is an already
          // declared class and it has appropriate
          // constructors and operators
  }
};
```

With the Initializer List, the following steps are followed by compiler:

1. Parameterised constructor of "Type" class is called to initialize: variable(a). The arguments in the initializer list are used to copy construct "variable" directly.

2. The destructor of "Type" is called for "a" since it goes out of scope.

As we can see from this example if we use assignment inside constructor body there are three function calls: **constructor + destructor + one addition assignment operator call**. And if we use Initializer List there are only two function calls: **copy constructor + destructor call**. See this post for a running example on this point. This assignment penalty will be much more in "real" applications where there will be many such variables. Thanks to ptr for adding this point.

6.10 Composition

In C++ Composition, an object is a part of another object. The object that is a part of another object is known as a sub-object. When a C++ Composition is destroyed, then all of its subobjects are destroyed as well. Such as when a car is destroyed, then its motor, frame, and other parts are also destroyed with it.

```
#include <string>
#include "Birthday.h"

class Person {
 public:
  Person(string n, Birthday b)
  : name(n),
   bd(b)
  {
  }
 private:
  string name;
  Birthday bd;
};
```

In the above code, the *Birthday* class, is used inside the *Person* class. As a summary, composition means adding/combining simple objects in order to create a bigger and more complex object.
One important note in the above example is that we wrote **#include <string>** and didn't say **.h** but for the *Birthday* library we used **#include "Birthday.h"** with " " and **.h**. The reason for this action is that the **string** class is a C++ standard library but the *Birthday* library is written by us in the same directory of our project.
In general, composition serves to keep each individual class relatively simple, straightforward, and focused on performing one task. It also enables each sub-object to be self-contained, allowing for reusability (we can use the Birthday class within various other classes).

6.11 Friend Functions

Normally, private members of a class cannot be accessed from outside of that class. However, declaring a non-member function as a **friend** of a class allows it to access the class' private members. This is accomplished by including a declaration of this external function within the class, and preceding it with the keyword **friend**. In the example below, *someFunc()*, which is not a member function of the class, is a friend of *MyClass* and can access its private members.

```

class MyClass {
 public:
  MyClass() {
   regVar = 0;
  }
 private:
  int regVar;

 friend void someFunc(MyClass &obj);
};

void someFunc(MyClass &obj) {
  obj.regVar = 42;
```

```
    cout << obj.regVar;
}
```

someFunc() had the ability to modify the private member of the object and print its value. Typical use cases of friend functions are operations that are conducted between two different classes accessing private members of both. You can declare a function friend across any number of classes. Similar to friend functions, you can define a friend class, which has access to the private members of another class.

6.12 This

Every object in C++ has access to its own address through an important pointer called the this pointer. Inside a member function this may be used to refer to the invoking object.
Friend functions do not have a this pointer, because friends are not members of a class.
The **printInfo()** method, in the code below, offers three alternatives for printing the member variable of the class. All three alternatives will produce the same result.

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

class MyClass {
    public:
        MyClass(int a) : var(a)
        { }
        void printInfo() {
            cout << var <<endl;
            cout << this->var <<endl;
            cout << (*this).var <<endl;
        }
    private:
        int var;
};

int main() {
    MyClass obj(42);
    obj.printInfo();
}
```

Output:
42
42
42

this is a **pointer** to the object, so the arrow selection operator is used to select the member variable. Note that only member functions have a this pointer.

You may be wondering why it's necessary to use the this keyword, when you have the option of directly specifying the variable. The this keyword has an important role in **operator overloading**.

6.13 Operator Overloading

Most of the C++ built-in operators can be redefined or overloaded. Thus, operators can be used with user-defined types as well (for example, allowing you to add two objects together). This chart shows the operators that can be overloaded:

+	-	*	/	%	^
&	|		!	,	=
<	<=	>	>=	++	–
«	»	==	!=	&&	||
+=	-=	*=	/=	%=	^=
&=	|=	«=	»=	[]	()
->	->*	new	new[]	delete	delete[]

Operators that can't be overloaded include ::, .*, ., ?:

Example:
Overloaded operators are functions, defined by the keyword **operator** followed by the symbol for the operator being defined. An overloaded operator is similar to other functions in that it has a **return type** and a **parameter list**.
In our example we will be overloading the **+ operator**. It will return an object of our class and take an object of our class as its parameter.

```
class MyClass {
 public:
  int var;
  MyClass() {}
  MyClass(int a)
  : var(a) { }

 // This is where we have overloaded the "+" operator.
  MyClass operator+(MyClass &obj) {
   MyClass res;
   res.var= this->var+obj.var;
   return res;
  }
};
int main() {
    MyClass obj1(12), obj2(55);
    MyClass res = obj1+obj2;

    cout << res.var;
}
```

Example:
We continue to develop our Queue management system that we made in the previous module. You are asked to add a new functionality: adding two queues together. The result should be a new queue, where the elements of the first queue come first, followed by the second queue's elements.
Given the Queue class, overload the + operator, so that the code in main works and successfully adds two queues.

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

class Queue {
  int size;
  int* queue;

  public:
  Queue() {
    size = 0;
    queue = new int[100];
  }
  void add(int data) {
    queue[size] = data;
    size++;
  }
  void remove() {
    if (size == 0) {
      cout << "Queue is empty"<<endl;
      return;
    }
    else {
      for (int i = 0; i < size - 1; i++) {
        queue[i] = queue[i + 1];
      }
      size--;
    }
  }
  void print() {
    if (size == 0) {
      cout << "Queue is empty"<<endl;
      return;
    }
    for (int i = 0; i < size; i++) {
      cout<<queue[i]<<" <- ";
    }
    cout << endl;
  }
```

```
40      //your code goes here
41      Queue operator +(Queue &q1){
42        Queue q3;
43        for (int i = 0 ; i < this->size ; i++){
44          q3.add(this->queue[i]);
45        }
46        for (int i = 0 ; i < q1.size ; i++){
47          q3.add(q1.queue[i]);
48        }
49            return q3;
50      }
51
52    };
53
54    int main() {
55      Queue q1;
56      q1.add(42); q1.add(2); q1.add(8);  q1.add(1);
57      Queue q2;
58      q2.add(3); q2.add(66); q2.add(128);  q2.add(5);
59      Queue q3 = q1+q2;
60      q3.print();
61
62      return 0;
63    }
```

Output:
42 <- 2 <- 8 <- 1 <- 3 <- 66 <- 128 <- 5 <-

7 Inheritance & Polymorphism

7.1 Inheritance

Inheritance is one of the most important concepts of object-oriented programming. Inheritance allows us to define a class based on another class. This facilitates greater ease in creating and maintaining an application. The class whose properties are inherited by another class is called the **Base** class. The class which inherits the properties is called the **Derived** class. The derived class inherits all feature from the base class, and can have its own additional features.
The idea of inheritance implements the **is-a** relationship. For example, mammal **IS-A** animal, dog IS-A mammal, hence dog IS-A animal as well.

```
1   class Mother
2   {
3    public:
4     Mother() {};
5     void sayHi() {
```

```
        cout << "Hi";
    }
};

class Daughter
{
 public:
  Daughter() {};
};
```

The Base class is specified using a **colon** and an access specifier public means, that all public members of the base class are public in the derived class. In other words, all public members of the Mother class become public members of the Daughter class.

```
class Daughter : public Mother
{
 public:
  Daughter() {};
};
```

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

class Mother
{
    public:
        Mother() {};
        void sayHi() {
            cout << "Hi";
        }
};

class Daughter: public Mother
{
    public:
        Daughter() {};
};

int main() {
    Daughter d;
    d.sayHi();
}
```

Output:
Hi

Attention

- A derived class inherits all base class methods with the following exceptions:
  - Constructors, Destructors
  - Overloaded operators
  - The friend functions

A class can be derived from multiple classes by specifying the base classes in a comma-separated list. For example: **class Daughter: public Mother, public Father{...};**

7.1.1 "Protected" Access Specifier

There is one more access specifier - **protected**. A protected member variable or function is very similar to a private member, with one difference - **it can be accessed in the derived classes**.

7.1.2 Type of Inheritance

Access specifiers are also used to specify the type of inheritance. Remember, we used public to inherit the Daughter class:
class Daughter: public Mother

private and *protected* access specifiers can also be used here.

- **Public** Inheritance: public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

- **Protected** Inheritance: public and protected members of the base class become protected members of the derived class.

- **Private** Inheritance: public and protected members of the base class become private members of the derived class.

Attention

Public inheritance is the most commonly used inheritance type. If no access specifier is used when inheriting classes, the type becomes **private** by default.

When inheriting classes, the base class' constructor and destructor are not inherited. However, they are being called when an object of the derived class is created or deleted. To further explain this behavior, let's create a sample class that includes a constructor and a destructor:

```
1  #include <iostream>
2  using namespace std;
3
4  class Mother {
5      public:
```

```
6           Mother()
7           {
8               cout <<"Mother Contructor"<<endl;
9           }
10          ~Mother()
11          {
12              cout <<"Mother Destructor"<<endl;
13          }
14  };
15
16
17  class Daughter: public Mother{
18
19      public:
20
21          Daughter(){
22              cout << "Doughter's Contructor"<<endl;
23          }
24
25          ~Daughter(){
26              cout << "Doughter's Destructor"<<endl;
27          }
28
29  };
30  int main() {
31      Daughter d;
32  }
```

Output:
Mother Contructor
Doughter's Contructor
Doughter's Destructor
Mother Destructor

Attention

- Note that the base class' constructor is called first, and the derived class' constructor is called next.

- When the object is destroyed, the derived class's destructor is called, and then the base class' destructor is called.

You can think of it as the following: The derived class needs its base class in order to work - that is why the base class is set up first.

7.2 Polymorphism

The word polymorphism means **"having many forms"**. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
C++ polymorphism means that a call to a member function will cause a different implementation to be executed depending on the type of object that invokes the function. **Simply, polymorphism means that a single function can have a number of different implementations.**

```
class Ninja: public Enemy {
 public:
  void attack() {
   cout << "Ninja! - "<<attackPower<<endl;
  }
};

class Monster: public Enemy {
 public:
  void attack() {
   cout << "Monster! - "<<attackPower<<endl;
  }
};
```

Ninja and Monster inherit from Enemy, so all Ninja and Monster objects are Enemy objects. This allows us to do the following:

```
Monster m;
Ninja n;
Enemy *e1 = &n;
Enemy *e2 = &m;
e1 -> attack();
e2 -> attack();
```

7.3 Virtual Functions

Continuing on with our game example, we want every Enemy to have an *attack()* function. To be able to call the corresponding *attack()* function for each of the derived classes using *Enemy* pointers, we need to declare the base class function as **virtual**. Defining a virtual function in the base class, with a corresponding version in a derived class, allows **polymorphism** to use Enemy pointers to call the derived classes' functions. Every derived class will **override** the attack() function and have a separate implementation:

```
class Enemy {
 public:
  virtual void attack() {
  // The derived classes will override this function.
```

```
    // we can also define a body for it in here.
    cout << "The Enemy Attacks" << endl;
    }
};

class Ninja: public Enemy {
 public:
  void attack() {
   cout << "Ninja!"<<endl;
  }
};

class Monster: public Enemy {
 public:
  void attack() {
   cout << "Monster!"<<endl;
 }
```

As the attack() function is declared virtual, it works like a **template**, telling that the derived class might have an attack() function of its own.
If a function in the base class is virtual, the function's implementation in the derived class is called according to the actual type of the object referred to, regardless of the declared type of the pointer. A class that declares or inherits a virtual function is called a **polymorphic** class.

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

class Enemy {
    public:
        virtual void attack() {
            cout << "Enemy!"<<endl;
        }
};

class Ninja: public Enemy {
    public:
        void attack() {
            cout << "Ninja!"<<endl;
        }
};

class Monster: public Enemy {
    public:
        void attack() {
            cout << "Monster!"<<endl;
        }
};

```

```
int main() {
    Ninja n;
    Monster m;
    Enemy e;

    Enemy *e1 = &n;
    Enemy *e2 = &m;
    Enemy *e3 = &e;

    e1->attack();
    // Outputs "Ninja!"

    e2->attack();
    // Outputs "Monster!"

    e3->attack();
    // Outputs "Enemy!"
}
```

Output:
Ninja!
Monster!
Enemy!

Attention

This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

7.4 Pure Virtual Functions

In some situations you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the functi
The virtual member functions without definition are known as **pure virtual functions**. They basically specify that the derived classes define that function on their own.
The syntax is to replace their definition by =0 (an equal sign and a zero):

```
class Enemy {
 public:
  virtual void attack() = 0;
};
```

The **= 0** tells the compiler that the function has no body.

Attention

- A **pure virtual function** basically defines, that the derived classes will have that function defined on their own.
- Every derived class inheriting from a class with a pure virtual function **must** override that function.
- If the pure virtual function is not overridden in the derived class, the code fails to compile and results in an error when you try to instantiate an object of the derived class.

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

class Enemy {
 public:
  virtual void attack() = 0;
};

class Ninja: public Enemy {
 public:
  void defend() {}
};

class Monster: public Enemy {
 public:
  void attack() {
   cout << "Monster!"<<endl;
  }
};

int main() {
  Ninja n;
  Monster m;

}
```

Output:
main.cpp: In function ’int main()’:
main.cpp:22:8: error: cannot declare variable ’n’ to be of abstract type ’Ninja’
Ninja n;
main.cpp:9:7: note: because the following virtual functions are pure within ’Ninja’:
class Ninja: public Enemy {
main.cpp:6:16: note: virtual void Enemy::attack()
virtual void attack() = 0;

7.4.1 Abstract Classes

You **cannot** create objects of the base class with a pure virtual function. Running the following code will return an error:

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

class Enemy {
    public:
        virtual void attack() = 0;
};

class Ninja: public Enemy {
    public:
        void attack() {
            cout << "Ninja!"<<endl;
        }
};

class Monster: public Enemy {
    public:
        void attack() {
            cout << "Monster!"<<endl;
        }
};


int main()
{
    Enemy e;
    return 0;
}
```

Output:
./Playground/file0.cpp: In function ’int main()’:
./Playground/file0.cpp:26:11: error: cannot declare variable ’e’ to be of abstract type ’Enemy’
26 | Enemy e;
|
./Playground/file0.cpp:4:7: note: because the following virtual functions are pure within ’Enemy’:
4 | class Enemy { |
./Playground/file0.cpp:6:22: note: ’virtual void Enemy::attack()’
6 | virtual void attack() = 0;
|

These classes are called **abstract**. They are classes that can only be used as base classes, and thus are allowed to have pure virtual functions. You might think that an abstract base class is useless, but it isn’t. It can be used to create pointers and take advantage of all its polymorphic abilities.

7.4.2 Reminding Some Tips

Attention

- Which keyword makes a class members to be accessible only to its derived classes? **answer:** Protected

- The derived class destructor is called before the base class destructor.

8 Templates

8.1 Function Templates

8.1.1 How to Define

To define a function template, use the keyword **template**, followed by the template type definition: **template <class T>**

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

template <class T>
T sum(T a, T b) {
    return a+b;
}

int main () {
    int x=7, y=15;
    cout << sum(x, y) << endl;
```

```
12      cout << sum(9.5, 10.75) << endl;
13  }
```

Output:
22
20.25

Attention

- When creating a template type parameter, the keyword **typename** may be used as an alternative to the keyword **class**:
  - **template <typename T>** instead of using **template <class T>**
- In this context, the keywords are identical, but throughout this course, we'll use the keyword class.
- Template functions can *save a lot of time* **(Time Efficiency)**, because they are *written only once*, and work with different types.
- Template functions reduce code **maintenance**, *because duplicate code is reduced significantly*.
- **Enhanced safety** is another advantage in using template functions, since *it's not necessary to manually copy functions and change types*.

8.1.2 Multiple Generic Data Types

Function templates also make it possible to work with multiple generic data types. Define the data types using a comma-separated list. Let's create a function that compares arguments of varying data types (an int and a double), and prints the smaller one.

```
1   #include <iostream>
2   using namespace std;
3
4   template <class T, class U>
5   T smaller(T a, U b) {
6       return (a < b ? a : b);
7   }
8
9   int main () {
10      int x=72;
```

```
    double y=15.34;
    cout << smaller(x, y) << endl;
}
/*The output converts to an integer, because we specified the
  function template's return type to be of the same type as the
  first parameter (T), which is an integer.*/
```

Output:
15

Attention

- Remember that when you declare a template parameter, you absolutely must use it in your function definition. Otherwise, the compiler will complain!

8.2 Class Templates

Just as we can define function templates, we can also define class templates, allowing classes to have members that use template parameters as types. The same syntax is used to define the class template:

```
template <class T>
class Pair {
 private:
  T first, second;
 public:
  Pair (T a, T b):
   first(a), second(b) {
  }
}
```

A specific syntax is required in case you define your member functions **outside of your class** - for example in a *separate source file*. You need to specify the generic type in angle brackets after the class name. For example, to have a member function bigger() defined outside of the class, the following syntax is used:

```
template <class T>
class Pair {
 private:
  T first, second;
 public:
  Pair (T a, T b):
   first(a), second(b){
  }
  T bigger();
};
```

```

template <class T>
T Pair<T>::bigger() {
  // some code
}
```

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

template <class T>
class Pair {
    private:
        T first, second;
    public:
        Pair (T a, T b):
        first(a), second(b) { }
        T bigger();
};

template <class T>
T Pair<T>::bigger() {
    return (first>second ? first : second);
}

int main()
{
    Pair <double> obj_dob(23.43, 5.68);
    Pair <int> obj_int(10,15);
    cout << obj_dob.bigger() << endl;
    cout << obj_int.bigger() << endl;
    return 0;
}
```

Output:
23.43
15

8.3 Template Specialization

In case of regular class templates, the way the class handles different data types is the same; the same code runs for all data types. **Template specialization** allows for the definition of a different implementation of a template when a specific type is passed as a template argument.

Problem

For example, we might need to handle the character data type in a different manner than we do numeric data types. To specify different behavior for the data type char, we would create a template specialization.

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

template <class T>
class MyClass {
    public:
        MyClass (T x) {
            cout <<x<<" -  not a char"<<endl;
        }
};

template < >
class MyClass<char> {
    public:
        MyClass (char x) {
            cout <<x<<" is a char!"<<endl;
        }
};

int main () {
    MyClass<int> ob1(42);
    MyClass<double> ob2(5.47);
    MyClass<char> ob3('s');
}
```

Output:
42 - not a char
5.47 - not a char
s is a char!

First of all, notice that we precede the class name with **template<>**, including an *empty parameter* list. This is because all types are known and no template arguments are required for this specialization, but still, it is the specialization of a class template, and thus it requires to be noted as such. But more important than this prefix, is the **<char>** specialization parameter after the class template name. **This specialization parameter itself identifies the type for which the template class is being specialized (char)**. In the example above, the first class is the generic template, while the second is the specialization. If necessary, your specialization can indicate a completely different behavior from the behavior of the generic template.

Attention

- Keep in mind that there is no member ”inheritance” from the generic template to the specialization, so all members of the template class specializations must be defined on their own.

9 Exceptions

C++ exception handling is built upon three keywords: **try, catch, and throw**.

- **throw:** is used to throw an exception when a problem shows up. In the throw statement, the operand determines a type for the exception. **This can be any expression**. The type of the expression's result will determine the type of the exception thrown.

```
int motherAge = 29;
int sonAge = 36;
if (sonAge > motherAge) {
  throw "Wrong age values";
}
```

- **try:** A try block identifies a block of code that will activate specific exceptions. It's followed by one or more catch blocks.
- **catch:** The catch keyword represents a block of code that executes when a particular exception is thrown.

Code that could generate an exception is surrounded with the *try/catch block*. You can specify what type of exception you want to catch by the exception.

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

int main()
{
    try {
        int motherAge = 29;
        int sonAge = 36;
        if (sonAge > motherAge) {
            throw 99;
        }
    }
    catch (int x) {
        cout<<"Wrong age values - Error "<<x;
    }

    return 0;
}
```

Output:
Wrong age values - Error 99

The *try block* throws the exception, and the *catch block* then handles it. The error code 99, which is an integer, appears in the throw statement, so it results in an exception of type int. **Multiple catch statements may be listed to handle various exceptions in case multiple exceptions are thrown by the try block.** It's possible to specify that your catch block handles any type of exception thrown in a try block. To accomplish this, add an **ellipsis (...)** between the parentheses of catch:

```
try {
  // code
} catch(...) {
  // code to handle exceptions
}
```

10 Working With Files

Another useful C++ feature is the ability to read and write to files. That requires the standard C++ library called **fstream**. Three new data types are defined in **fstream**:

- **ofstream:** Output file stream that creates and writes information to files.
- **ifstream:** Input file stream that reads information from files.
- **fstream:** General file stream, with both ofstream and ifstream capabilities that allow it to create, read, and write information to files.

```
#include <iostream>
#include <fstream>
```

Attention

- To perform file processing in C++, header files **<iostream>** and **<fstream>** must be included in the C++ source file.
- These classes are derived directly or indirectly from the classes **istream** and **ostream**. We have already used objects whose types were these classes: **cin** is an object of class *istream* and **cout** is an object of class *ostream*.

10.1 Sample

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream** object may be used to open a file for writing.
Let's open a file called "test.txt" and write some content to it:

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

int main() {
  ofstream MyFile;
  MyFile.open("test.txt");

  MyFile << "Some text. \n";
  MyFile.close();
}
```

The above code creates an *ofstream* object called MyFile, and uses the open() function to open the *"test.txt"* file on the file system. As you can see, the same stream output operator is used to write into the file. If the specified file does not exist, the open function will create it automatically. You also have the option of specifying a path for your file in the open function, since it can be in a location other than that of your project. When you've finished working with a file, close it using the member function **close()**.
You can also provide the path to your file using the *ofstream* objects constructor, instead of calling the open function.

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

int main() {
  ofstream MyFile("test.txt");

  MyFile << "This is awesome! \n";
  MyFile.close();
}
```

. Under certain circumstances, such as when you don't have file permissions, the open function can fail. The **is_open()** member function checks whether the **file is open and ready to be accessed**.

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

int main() {
  ofstream MyFile("test.txt");

  if (MyFile.is_open()) {
   MyFile << "This is awesome!  The file is ready/open\n";
  }
  else {
   cout << "Something went wrong";
```

```
    }
    MyFile.close();
}
```

An optional second parameter of the open function defines the **mode** in which the file is opened. This list shows the supported modes.

Mode Paramater	Meaning
ios::app	Append to the end of file
ios::ate	Go the end of file while opening
ios::binary	Open file in binary mode
ios::in	Open file in read_only mode
ios::out	Open file for writing only
ios:: trunc	Delete the contents of the file if it exists

All these flags can be combined using the bitwise operator **OR (|)**.

Problem

For example, to open a file in write mode and truncate it, in case it already exists, use the following syntax:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Problem

Read all contents of a file

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

int main () {
    ofstream MyFile1("test.txt");

    MyFile1 << "This is awesome! \n";
    MyFile1.close();

    string line;
    ifstream MyFile("test.txt");
    while ( getline (MyFile, line) ) {
        cout << line << '\n';
    }
    MyFile.close();
}
```

The **getline()** function reads characters from an input stream and places them into a string.

11

11.1 Sample

11.1.1 Sample

Problem

Output:

Attention

-