Template in C++

**Generics in C++ =>**

Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.

The method of Generic Programming is implemented to increase the efficiency of the code.

Generic Programming enables the programmer to write a general algorithm which will work with all data types.

It eliminates the need to create different algorithms if the data type is an integer, string or a character.

*Generics can be implemented in C++ using*[*Templates*](https://www.geeksforgeeks.org/templates-cpp/)*.*

Template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don’t need to write the same code for different data types.

### **Generic Functions using Template:**

We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

#include <iostream>

using namespace std;

// One function works for all data types.

// This would work even for user defined types

// if operator '>' is overloaded

template <typename T>

T myMax(T x, T y) //replace the data\_type with T everywhere.

{

return (x > y) ? x : y;

}

int main()

{

// Call myMax for int

cout << myMax<int>(3, 7) << endl;

// call myMax for double

cout << myMax<double>(3.0, 7.0) << endl;

// call myMax for char

cout << myMax<char>('g', 'e') << endl;

return 0;

}

### **Generic Class using Template:**

Like function templates, class templates are useful when a class defines something that is independent of data type. Can be useful for classes like LinkedList, binary tree, Stack, Queue, Array, etc.

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  template <typename T>  class Array {  private:  T\* ptr;  int size;  public:  Array(T arr[], int s);  void print();  };  template <typename T>  Array<T>::Array(T arr[], int s)  {  ptr = new T[s];  size = s;  for (int i = 0; i < size; i++)  ptr[i] = arr[i];  }  template <typename T>  void Array<T>::print()  {  for (int i = 0; i < size; i++)  cout << " " << \*(ptr + i);  cout << endl;  } | int main()  {  int arr[5] = { 1, 2, 3, 4, 5 };  Array<int> a(arr, 5);  a.print();  return 0;  } |

### **Working with multi-type Generics:**

We can pass more than one data types as arguments to templates. The following example demonstrates the same.

|  |
| --- |
| #include <iostream>  using namespace std;  template <class T, class U>  class A {  T x;  U y;  public:  A()  {  cout << "Constructor Called" << endl;  }  };  int main()  {  A<char, char> a;  A<int, double> b;  return 0;  } |

# Const Qualifier in C

The qualifier **const**can be applied to the declaration of any variable to specify that its value will not be changed (which depends upon where const variables are stored, we may change the value of the const variable by using a pointer).

Using the const qualifier in C is a good practice when we want to ensure that some values should remain constant and should not be accidentally modified.

### 1. Constant Variables

const int var = 100;

In this case, const is used to declare a variable **var**as a constant with an initial value of 100. The value of this variable cannot be modified once it is initialized.

|  |
| --- |
| #include <stdio.h>    **int** main()  {  **const** **int** var = 100;        // Compilation error: assignment of read-only variable      // 'var'      var = 200;    **return** 0;  } |

**Output**

./Solution.cpp: In function 'int main()':

./Solution.cpp:11:9: error: assignment of read-only variable 'var'

var = 200;

### 2. Pointer to Constant

const int\* ptr;

OR

int const \*ptr;

We can change the pointer to point to any other integer variable, but cannot change the value of the object (entity) pointed using pointer ptr.

The pointer is stored in the read-write area (stack in the present case). The object pointed may be in the read-only or read-write area.

|  |
| --- |
| #include <stdio.h>  **int** main(**void**)  {  **int** i = 10;  **int** j = 20;      /\* ptr is pointer to constant \*/  **const** **int**\* ptr = &i;    **printf**("ptr: %d\n", \*ptr);      /\* error: object pointed cannot be modified      using the pointer ptr \*/      \*ptr = 100;        ptr = &j; /\* valid \*/  **printf**("ptr: %d\n", \*ptr);    **return** 0;  } |

**Output:-**

./Solution.c: In function 'main':

./Solution.c:12:10: error: assignment of read-only location '\*ptr'

\*ptr = 100;

### 3. Constant Pointer to Variable

int\* const ptr;

The above declaration is a constant pointer to an integer variable, which means we can change the value of the object pointed by the pointer, but cannot change the pointer to point to another variable.

|  |
| --- |
| #include <stdio.h>    **int** main(**void**)  {  **int** i = 10;  **int** j = 20;      /\* constant pointer to integer \*/  **int**\* **const** ptr = &i;    **printf**("ptr: %d\n", \*ptr);        \*ptr = 100; /\* valid \*/  **printf**("ptr: %d\n", \*ptr);        ptr = &j; /\* error \*/  **return** 0;  } |

**Output**

./Solution.c: In function 'main':

./Solution.c:15:9: error: assignment of read-only variable 'ptr'

ptr = &j; /\* error \*/

### 4. Constant Pointer to Constant

const int\* const ptr;

The above declaration is a constant pointer to a constant variable which means we cannot change the value pointed by the pointer as well as we cannot point the pointer to other variables.

|  |
| --- |
| #include <stdio.h>    **int** main(**void**)  {  **int** i = 10;  **int** j = 20;      /\* constant pointer to constant integer \*/  **const** **int**\* **const** ptr = &i;    **printf**("ptr: %d\n", \*ptr);        ptr = &j; /\* error \*/      \*ptr = 100; /\* error \*/    **return** 0;  } |

**Output**

./Solution.c: In function 'main':

./Solution.c:12:9: error: assignment of read-only variable 'ptr'

ptr = &j; /\* error \*/

^

./Solution.c:13:10: error: assignment of read-only location '\*ptr'

\*ptr = 100; /\* error \*/

## **Preprocessor Directives**

Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling.

All of these preprocessor directives begin with a ‘#’ (hash) symbol.

The ‘#’ symbol indicates that whatever statement starts with a ‘#’ will go to the preprocessor program to get executed.

We can place these preprocessor directives anywhere in our program.

### ***List of preprocessor directives in C :-***

|  |  |
| --- | --- |
| **Preprocessor Directives** | **Description** |
| **#define** | Used to define a macro |
| **#undef** | Used to undefine a macro |
| **#include** | Used to include a file in the source code program |
| **#ifdef** | Used to include a section of code if a certain macro is defined by #define |
| **#ifndef** | Used to include a section of code if a certain macro is not defined by #define |
| **#if** | Check for the specified condition |
| **#else** | Alternate code that executes when #if fails |
| **#endif** | Used to mark the end of #if, #ifdef, and #ifndef |

## Types of C Preprocessors:-

**There are 4 Main Types of Preprocessor Directives:**

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

## **1. Macros**

In C, Macros are pieces of code in a program that is given some name.

Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code.

### Syntax of Macro Definition

**#define** *token* *value*

### Macros With Arguments

We can also pass arguments to macros. Macros defined with arguments work similarly to functions.

**#define** foo(*a, b*) *a + b*  
#define func(r) r \* r

|  |
| --- |
| // macro with parameter  #define AREA(l, b) (l \* b)    **int** main()  {  **int** l1 = 10, l2 = 5, area;        area = AREA(l1, l2);    **printf**("Area of rectangle is: %d", area);    **return** 0;  } |

## **2.** **File Inclusion**

This type of preprocessor directive tells the compiler to include a file in the source code program.

The **#include preprocessor directive** is used to include the header files in the C program.

**There are two types of files that can be included by the user in the program:**

### **Standard Header Files**

The standard header files contain definitions of pre-defined functions like **printf(), scanf(),**etc.

**Syntax**

**#include** <*file\_name*>

### **User-defined Header Files**

When a program becomes very large, it is a good practice to divide it into smaller files and include them whenever needed.

**#include** "*filename*"

## **3. Conditional Compilation**

 type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions.

 There are the following preprocessor directives that are used to insert conditional code:

1. **#if Directive**
2. **#ifdef Directive**
3. **#ifndef Directive**
4. **#else Directive**
5. **#elif Directive**
6. **#endif Directive**

**Syntax**

**#ifdef** *macro\_name*  
 // Code to be executed if macro\_name is defined

#**ifndef** *macro\_name*  
 // Code to be executed if macro\_name is not defined

**#if** *constant\_expr*  
 // Code to be executed if constant\_expression is true

**#elif** *another\_constant\_expr*  
 // Code to be excuted if another\_constant\_expression is true

**#else**  
 // Code to be excuted if none of the above conditions are true  
**#endif**

|  |  |
| --- | --- |
| #ifdef MACRO  controlled text  #endif /\* macroname \*/ | ifndef macro\_name  statement1;  statement2;  statement3;  .  statementN;  endif |
| #if macro\_condition  statements  #elif macro\_condition  statements  #else  statements  #endif |  |

## 4. Other Directives

Apart from the above directives, there are two more directives that are not commonly used. These are:

1. **#undef Directive**
2. **#pragma Directive**

### **1. #undef Directive**

The #undef directive is used to undefine an existing macro.

#undef MacroName

### 2. #pragma Directive

#pragma *directive*

1. **#pragma startup:**These directives help us to specify the functions that are needed to run before program startup (before the control passes to main())
2. **#pragma exit**: These directives help us to specify the functions that are needed to run just before the program exit (just before the control returns from main()).

# Namespace in C++

* Namespace provide the space where we can define or declare identifier i.e. variable,  method, classes.
* Using namespace, you can define the space or context in which identifiers are defined i.e. variable, method, classes. In essence, a namespace defines a scope.

### Defining a Namespace:

* A namespace definition begins with the keyword namespace followed by the namespace name as follows:

|  |
| --- |
| * namespace namespace\_name * { * // code declarations i.e. variable (int a;) * method (void add();) * classes ( class student{};) * } |

* To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:
  + **namespace\_name: :code;**  // code could be variable , function or class.

### The using directive:

You can also avoid prepending of namespaces with the using namespace directive.

This directive tells the compiler that the subsequent code is making use of names in the specified namespace.

|  |
| --- |
| #include <iostream>  using namespace std;  // first name space  namespace first\_space  {  void func()  {  cout << "Inside first\_space" << endl;  }  }  // second name space  namespace second\_space  {  void func()  {  cout << "Inside second\_space" << endl;  }  }  using namespace first\_space;  int main ()  {  // This calls function from first name space.  func();  return 0;  } |

**Output**

Inside first\_space

Names introduced in a using directive obey normal scope rules. The name is visible from the point of the using directive to the end of the scope in which the directive is found.

### Nested Namespaces:

* Namespaces can be nested where you can define one namespace inside another name space as follows:

|  |
| --- |
| * namespace namespace\_name1 * { * // code declarations * namespace namespace\_name2 * { * // code declarations * } * } |

You can access members of nested namespace by using resolution operators as follows:  
// to access members of namespace\_name2  
using namespace namespace\_name1::namespace\_name2;

// to access members of namespace\_name1  
using namespace namespace\_name1;

# Destructors in C++

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed.

Meaning, a destructor is the last function that is going to be called before an object is destroyed.

* A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
* Destructor has the same name as their class name preceded by a tilde (~) symbol.
* It is not possible to define more than one destructor.
* The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded.
* Destructor neither requires any argument nor returns any value.
* It is automatically called when an object goes out of scope.
* Destructor release memory space occupied by the objects created by the constructor.
* In destructor, objects are destroyed in the reverse of an object creation.

Syntax for defining the destructor within the class:

~ <class-name>() {

*// some instructions*

}

syntax for defining the destructor outside the class:

<class-name> :: ~<class-name>() {

// some instructions

}

|  |
| --- |
| #include<iostream>  using namespace std;  class student {  private:  int a;    public:  int \*b;  student(){  a=5;  b=new int;  \*b=a;  }    ~student(){  free(b);  printf("\nfreed");  }  };  int main(){  class student s1;    printf("%d",\*(s1.b));    return 0;    } |

**Destructor in inheritance**

First the destructor of base class is called then destructor of virtual class is called.

|  |
| --- |
|  |