Template

A 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. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.   
C++ adds two new keywords to support templates: *‘template’*and *‘typename’*. The second keyword can always be replaced by keyword ‘class’.

**How do templates work?**   
Templates are expanded at compiler time. This is like macros. The difference is, the compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| #include <iostream>  using namespace std;  template <typename T>  T myMax(T x, T y)  {return (x > y)? x: y;}   int main()  {    cout << myMax<int>(3, 7) << endl;  // Call myMax for int    cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double    cout << myMax<char>('g', 'e') << endl;   // call myMax for char      return 0;  }  can there be more then one template in class   |  | | --- | | #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;  } |   Output:  Constructor Called  Constructor Called     |  | | --- | | #include<iostream>  using namespace std;  template<class T, class U = char>  class A  {  public:      T x;      U y;      A() {   cout<<"Constructor Called"<<endl;   }  };    int main()  {     A<char> a;  // This will call A<char, char>     return 0;  } |   Output:  Constructor Called   1. #include <iostream> 2. **using** **namespace** std; 3. **template**<**class** T> 4. **class** A 5. { 6. **public**: 7. T num1 = 5; 8. T num2 = 6; 9. **void** add() 10. { 11. std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl; 12. } 14. }; 16. **int** main() 17. { 18. A<**int**> d; 19. d.add(); 20. **return** 0; 21. }   function templates and static variables in c++   |  | | --- | | #include <iostream>  using namespace std;   template <typename T>  void fun(const T& x)  {    static int i = 10;    cout << ++i;    return 0;  }  int main()  {    fun<int>(1);  // prints 11    cout << endl;    fun<int>(2);  // prints 12    cout << endl;    fun<double>(1.1); // prints 11    cout << endl;    getchar();    return 0;  } |   Output of the above program is:  11  12  11  **What is the difference between function overloading and templates?**  Both function overloading and templates are examples of polymorphism feature of OOP. Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations.  **Can we pass nontype parameters to templates?**  We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template. The important thing to note about non-type parameters is, they must be const.  #include <iostream>  using namespace std;    template <class T, int max>  int arrMin(T arr[], int n)  {     int m = max;     for (int i = 0; i < n; i++)        if (arr[i] < m)           m = arr[i];       return m;  }    int main()  {     int arr1[]  = {10, 20, 15, 12};     int n1 = sizeof(arr1)/sizeof(arr1[0]);       char arr2[] = {1, 2, 3};     int n2 = sizeof(arr2)/sizeof(arr2[0]);       // Second template parameter to arrMin must be a constant     cout << arrMin<int, 10000>(arr1, n1) << endl;     cout << arrMin<char, 256>(arr2, n2);     return 0;  }  Template specialization:  [Template in C++](https://www.geeksforgeeks.org/templates-cpp/)is a feature. We write code once and use it for any data type including user defined data types. For example, sort() can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.  *What if we want a different code for a particular data type?* Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when sort() is called for char data type?  *It is possible in C++ to get a special behavior for a particular data type. This is called template specialization*.   |  | | --- | | #include <iostream>  using namespace std;  template <class T>  void fun(T a)  {     cout << "The main template fun(): "          << a << endl;  template<>  void fun(int a)  {      cout << "Specialized Template for int type: "           << a << endl;  }    int main()  {      fun<char>('a');      fun<int>(10);      fun<float>(10.14);  } |   Output:  The main template fun(): a  Specialized Template for int type: 10  The main template fun(): 10.14 |
|  |