**Function templates.**

Overloaded functions may have the same definition. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // overloaded functions  #include <iostream>  using namespace std;  int sum (int a, int b)  {  return a+b;  }  double sum (double a, double b)  {  return a+b;  }  int main ()  {  cout << sum (10, 20) << '\n';  cout << sum (1.0, 1.5) << '\n';  return 0;  } | 30  2.5 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions2/) |

Here, *sum* is overloaded with different parameter types, but with the exact same body.

The function *sum* could be overloaded for a lot of types, and it could make sense for all of them to have the same body. For cases such as this, C++ has the ability to define functions with generic types, known as *function templates*. Defining a function template follows the same syntax as a regular function, except that it is preceded by the template keyword and a series of template parameters enclosed in angle-brackets <>:

*template <template-parameters> function-declaration*

The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the *class* or *typename* keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | template <class SomeType>  SomeType sum (SomeType a, SomeType b)  {  return a+b;  } |  |

It makes no difference whether the generic type is specified with keyword *class* or keyword *typename* in the template argument list (they are 100% synonyms in template declarations).

In the code above, declaring *SomeType* (a generic type within the template parameters enclosed in angle-brackets) allows *SomeType* to be used anywhere in the function definition, just as any other type; it can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.

Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the *function template*, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:

*name <template-arguments> (function-arguments)*

For example, the sum function template defined above can be called with:

|  |  |  |
| --- | --- | --- |
|  | x = sum<int>(10,20); |  |

The function *sum<int>* is just one of the possible instantiations of function template *sum*. In this case, by using *int* as template argument in the call, the compiler automatically instantiates a version of *sum* where each occurrence of *SomeType* is replaced by *int*, as if it was defined as:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | int sum (int a, int b)  {  return a+b;  } |  |

Let's see an actual example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // function template  #include <iostream>  using namespace std;  template <class T>  T sum (T a, T b)  {  T result;  result = a + b;  return result;  }  int main () {  int i=1, j=7, k;  double f=8.0, g=0.1, h;  k=sum<int>(i,j);  h=sum<double>(f,g);  cout << k << '\n';  cout << h << '\n';  return 0;  } | 8  8.1 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions2/) |

In this case, we have used *T* as the template parameter name, instead of *SomeType*. It makes no difference, and *T* is actually a quite common template parameter name for generic types.  
In the example above, we used the function template *sum* twice. The first time with arguments of type *int*, and the second one with arguments of type *double*. The compiler has instantiated and then called each time the appropriate version of the function.

Note also how *T* is also used to declare a local variable of that (generic) type within *sum*:

|  |  |  |
| --- | --- | --- |
|  | T result; |  |

Therefore, result will be a variable of the same type as the parameters *a* and *b*, and as the type returned by the function. In this specific case where the generic type *T* is used as a parameter for *sum*, the compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets. Therefore, instead of explicitly specifying the template arguments with:

|  |  |  |
| --- | --- | --- |
| 1 2 | k = sum<int> (i,j);  h = sum<double> (f,g); |  |

It is possible to instead simply write:

|  |  |  |
| --- | --- | --- |
| 1 2 | k = sum (i,j);  h = sum (f,g); |  |

without the type enclosed in angle brackets. Naturally, for that, the type shall be unambiguous. If sum is called with arguments of different types, the compiler may not be able to deduce the type of *T* automatically.

Another program where generic type is specified with keyword *typename* instead keyword *class* in the template argument list. It makes no difference anyways.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | #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 large(T x, T y)  {  return (x > y) ? x : y;  }  int main()  {  cout << large<int>(8, 1) << endl; //call large for int  cout << large<double>(0.1, 8.8) << endl; //call large for double  cout << large<char>('h', 'k') << endl; //call large for char  return 0;  } | 8  8.8  k | [Edit & Run](https://cplusplus.com/doc/tutorial/functions2/) |

Templates are a powerful and versatile feature. They can have multiple template parameters, and the function can still use regular non-templated types. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // function templates  #include <iostream>  using namespace std;  template <class T, typename U>  bool are\_equal (T a, U b)  {  return (a == b);  }  int main ()  {  if (are\_equal(8, 8.0))  cout << "x and y are equal\n";  else  cout << "x and y are not equal\n";  return 0;  } | x and y are equal | [Edit & Run](https://cplusplus.com/doc/tutorial/functions2/) |

Note that this example uses automatic template parameter deduction in the call to *are\_equal*:

|  |  |  |
| --- | --- | --- |
|  | are\_equal(8, 8.0) |  |

Is equivalent to:

|  |  |  |
| --- | --- | --- |
|  | are\_equal<int,double>(8, 8.0) |  |

Since, in C++, integer literals with no suffix (such as 8) are always of type *int*, and floating-point literals without suffix (such as 8.0) are always of type *double*, there is no ambiguity possible, and thus the template arguments can be omitted in the call.

There is no ambiguity possible because numerical literals are always of a specific type: Unless otherwise specified with a suffix, integer literals always produce values of type *int*, and floating-point literals always produce values of type *double*. Therefore 8 has always type *int* and 8.0 has always type *double*.