Template parameters and template arguments

Every template is parametrized by one or more template parameters, indicated in the *parameter-list* of the template declaration syntax:

template < *parameter-list* **>** *declaration*	(1)

Each parameter in *parameter-list* may be one of

- non-type template parameter
- type template parameter
- template template parameter

Non-type template parameter

type name(optional)	(1)	
type name(optional) **=** *default*	(2)	
type **...** *name*(optional)	(3)	(since C++11)

1) A non-type template parameter with an optional name
2) A non-type template parameter with an optional name and a default value
3) A non-type template parameter pack with an optional name

type is one of the following types (optionally cv-qualified, the qualifiers are ignored)

- integral type
- enumeration
- pointer to object or to function
- lvalue reference to object or to function
- pointer to member object or to member function
- `std::nullptr_t` (since C++11)

Array and function types may be written in a template declaration, but they are automatically replaced by pointer to data and pointer to function as appropriate.

When the name of a non-type template parameter is used in an expression within the body of the class template, it is an unmodifiable prvalue unless its type was an lvalue reference type.

A template parameter of the form `class Foo` is not an unnamed non-type template parameter of type Foo, even if otherwise `class Foo` is an elaborated type specifier and `class Foo x;` declares x to be of type Foo.

Type template parameter

typename *name*(optional)	(1)	
class *name*(optional)	(2)	
typename|class *name*(optional) **=** *default*	(3)	
typename|class ... *name*(optional)	(4)	(since C++11)

1) A type template parameter with an optional name
2) Exactly the same as 1)
3) A type template parameter with an optional name and a default
4) A type template parameter pack with an optional name

In the body of the template declaration, the name of a type parameter is a typedef-name which aliases the type supplied when the template is instantiated.

There is no difference between the keywords **class** and **typename** in a type template parameter declaration.

Template template parameter

template < *parameter-list* **> typename(C++17)|class** *name*(optional)	(1)	
template < *parameter-list* **> typename(C++17)|class** *name*(optional) **=** *default*	(2)	
template < *parameter-list* **> typename(C++17)|class ...** *name*(optional)	(3)	(since C++11)

1) A template template parameter with an optional name
2) A template template parameter with an optional name and a default

3) A template template parameter pack with an optional name

Unlike type template parameter declaration, template template parameter declaration can only use the keyword **class** and not **typename**. (until C++17)

In the body of the template declaration, the name of this parameter is a template-name (and needs arguments to be instantiated)

```
template<class T> class myarray {};

// two type template parameters and one template template parameter:
template<class K, class V, template<typename> class C = myarray>
class Map {
  C<K> key;
  C<V> value;
};
```

Template arguments

In order for a template to be instantiated, every template parameter (type, non-type, or template) must be replaced by a corresponding template argument. For class templates, the arguments are either explicitly provided or defaulted. For function templates, the arguments are explicitly provided, defaulted, or deduced from context.

Template non-type arguments

The following limitations apply when instantiating templates that have non-type template parameters:

- For integral and arithmetic types, the template argument provided during instantiation must be a converted constant expression of the template parameter's type (so certain implicit conversion applies).
- For pointers to objects, the template arguments have to designate the address of an object with static storage duration and a linkage (either internal or external), or a constant expression that evaluates to the appropriate null pointer or `std::nullptr_t` value.
- For pointers to functions, the valid arguments are pointers to functions with linkage (or constant expressions that evaluate to null pointer values).
- For lvalue reference parameters, the argument provided at instantiation cannot be a temporary, an unnamed lvalue, or a named lvalue with no linkage (in other words, the argument must have linkage). (until C++17)
- For pointers to members, the argument has to be a pointer to member expressed as `&Class::Member` or a constant expression that evaluates to null pointer or `std::nullptr_t` value.

In particular, this implies that string literals, addresses of array elements, and addresses of non-static members cannot be used as template arguments to instantiate templates whose corresponding non-type template parameters are pointers to objects.

The template argument that can be used with a non-type template parameter can be any converted constant expression of the type of the template parameter.

```
template<const int* pci> struct X {};
int ai[10];
X<ai> xi;  // OK: array to pointer conversion and qualification conversions

struct Y {};
template<const Y& b> struct Z {};
Y y;
Z<y> z;  // OK: no conversion, extra cv-qualification is okay

template<int (&pa)[5]> struct W {};
int b[5];
W<b> w; // OK: no conversion

void f(char);
void f(int);
template<void (*pf)(int)> struct A {};
A<&f> a; // OK: overload resolution selects f(int)
```

The only exceptions are that non-type template parameters of *reference* and *pointer* type cannot refer to/be the address of (since C++17)

- a subobject (including non-static class member, base subobject, or array element)
- a temporary object (including one created during reference initialization)
- a string literal
- the result of typeid
- or the predefined variable `__func__`.

```
template<class T, const char* p> class X {};
X<int, "Studebaker"> x1; // Error: string literal as template-argument

template<int* p> class X {};
int a[10];
```

```
struct S { int m; static int s; } s;
X<&a[2]> x3;  // error: address of array element
X<&s.m> x4;   // error: address of non-static member
X<&s.s> x5;   // OK: address of static member
X<&S::s> x6;  // also OK: address of static member

template<const int& CRI> struct B { /* ... */ };
B<1> b2;     // error: temporary would be required for template argument
int c = 1;
B<c> b1;     // OK
```

Template type arguments

A template argument for a type template parameter must be a type-id, which may name an incomplete type:

```
template <class T> class X { }; // class template

struct A; // incomplete type
typedef struct {} B; // type alias to an unnamed type
int main() {
   X<A> x1; // OK, 'A' names a type
   X<A*> x2; // OK, 'A*' names a type
   X<B> x3; // OK, 'B' names a type
}
```

Template template arguments

A template argument for a template template parameter must be an id-expression which names a class template or a template alias.

When the argument is a class template, only the primary template is considered when matching the parameter. The partial specializations, if any, are only considered when a specialization based on this template template parameter happens to be instantiated.

```
template<class T> class A { // primary template
    int x;
};
template<class T> class A<T*> { // partial specialization
    long x;
};

// class template with a template template parameter V
template< template<typename> class V> class C {
    V<int> y; // uses the primary template
    V<int*> z; // uses the partial specialization
};

C<A> c; // c.y.x has type int, c.z.x has type long
```

To match a template template argument `A` to a template template parameter `P`, each of the template parameters of `A` must match corresponding template parameters of `P`. If `P`'s parameter list includes a parameter pack, zero or more template parameters (or parameter packs) from `A`'s template parameter list are matched by it.

```
template<class T> struct eval; // primary template

template< template<class, class...> class TT, class T1, class... Rest>
struct eval<TT<T1, Rest...>> {}; // partial specialization of eval

template <class T1> struct A;
template <class T1, class T2> struct B;
template <int N> struct C;
template <class T1, int N> struct D;
template <class T1, class T2, int N = 17> struct E;

eval<A<int>> eA; // OK: matches partial specialization of eval
eval<B<int, float>> eB; // OK: matches partial specialization of eval
eval<C<17>> eC; // error: C does not match TT in partial specialization because
                // TT's first parameter is a type template parameter,
                // while 17 does not name a type
eval<D<int, 17>> eD; // error: D does not match TT in partial specialization
                     // because TT's second parameter is a type parameter pack,
                     // while 17 does not name a type
eval<E<int, float>> eE; // error: E does not match TT in partial specialization
                        // because E's third (default) parameter is a non-type
```

Default template arguments

Default template arguments are specified in the parameter lists after the `=` sign. Defaults can be specified for any kind of template parameter (type, non-type, or template), but not to parameter packs.

If the default is specified for a template parameter of a primary class template, each subsequent template parameter must have a default argument, except the very last one may be a template parameter pack. In a function template, a parameter pack may be followed by more type parameters only if they have defaults or can be deduced from the function arguments.

Default parameters are not allowed

- in the out-of-class definition of a member template (they have to be provided in the declaration inside the class body)
- in friend class template declarations

On a friend function template declaration, default template arguments are allowed only if the declaration is a definition, and no other declarations of this function appear in this translation unit.

Default template arguments that appear in the declarations and the definition are merged similarly to default function arguments:

```
template<class T1, class T2 = int> class A;
template<class T1 = int, class T2> class A;
// the above is the same as the following:
template<class T1 = int, class T2 = int> class A;
```

But the same parameter cannot be given default arguments twice in the same scope

```
template<class T = int> class X;
template<class T = int> class X { /* ... */ }; // error
```

The template parameter lists of template template parameters can have their own default arguments, which are only in effect where the template template parameter itself is in scope:

```
// class template, with a type template parameter with a default
template <class T = float> struct B {};

// template template parameter T has a parameter list, which
// consists of one type template parameter with a default
template <template <class = float> class T> struct A {
    void f();
    void g();
};
// out-of-body member function template definitions
template <template <class TT> class T>
void A<T>::f() {
    T<> t; // error - TT has no default in scope
}
template <template <class TT = char> class T>
void A<T>::g() {
    T<> t; // OK, t is T<char>
}
```

Member access for the names used in a default template parameter is checked at the declaration, not at the point of use:

```
class B {};

template <class T> class C {
 protected:
    typedef T TT;
};

template <class U, class V = typename U::TT>
class D : public U {};

D<C<B>>* d; // access error, C::TT is protected
```

Examples

Non-type template parameters

Run this code

```
#include <iostream>

// simple non-type template parameter
template<int N>
```

```
struct S {
    int a[N];
};

template<const char*>
struct S2 {};

// complicated non-type example
template <
    char c, // integral type
    int (&ra)[5], // lvalue reference to object (of array type)
    int (*pf)(int), // pointer to function
    int (S<10>::*a)[10] // pointer to member object (of type int[10])
> struct Complicated {
    // calls the function selected at compile time
    // and stores the result in the array selected at compile time
    void foo(char base) {
        ra[4] = pf(c - base);
    }
};

//  S2<"fail"> s2; // Error: string literal cannot be used
char okay[] = "okay"; // static object with linkage
// S2< &okay[0] > s2; // Error: array element has no linkage
S2<okay> s2; // works

int a[5];
int f(int n) { return n;}
int main()
{
    S<10> s; // s.a is an array of 10 int
    s.a[9] = 4;

    Complicated<'2', a, f, &S<10>::a> c;
    c.foo('0');

    std::cout << s.a[9] << a[4] << '\n';
}
```

Output:

```
42
```

This section is incomplete
Reason: more examples

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/language/template_parameters&oldid=77806"