
3.2. Old "New" Language Features

Although C++98 is more than 10 years old now, programmers still can be surprised by some of the language features. Some of those are presented in this section.

Nontype Template Parameters

In addition to type parameters, it is also possible to use nontype parameters. A nontype parameter is then considered part of the type. For example, for the standard class `bitset<>` (see Section 12.5, page 650), you can pass the number of bits as the template argument. The following statements define two bitfields: one with 32 bits and one with 50 bits:

```
bitset<32> flags32;     // bitset with 32 bits
bitset<50> flags50;     // bitset with 50 bits
```

These bitsets have different types because they use different template arguments. Thus, you can't assign or compare them unless a corresponding type conversion is provided.

Default Template Parameters

Class templates may have default arguments. For example, the following declaration allows one to declare objects of class `MyClass` with one or two template arguments:

```
template <typename T, typename container = vector<T>>
class MyClass;
```

If you pass only one argument, the default parameter is used as the second argument:

```
MyClass<int> x1;          // equivalent to: MyClass<int,vector<int>>
```

Note that default template arguments may be defined in terms of previous arguments.

Keyword `typename`

The keyword `typename` was introduced to specify that the identifier that follows is a type. Consider the following example:

```
template <typename T>
class MyClass {
   typename T::SubType * ptr;
   ...
};
```

Here, `typename` is used to clarify that `SubType` is a type defined within class `T`. Thus, `ptr` is a pointer to the type `T::SubType`. Without `typename`, `SubType` would be considered a static member, and thus

```
T::SubType * ptr
```

would be a multiplication of value `SubType` of type `T` with `ptr`.

According to the qualification of `SubType` being a type, any type that is used in place of `T` must provide an inner type `SubType`. For example, the use of type `Q` as a template argument is possible only if type `Q` has an inner type definition for `SubType`:

```
class Q {
    typedef int SubType;
    ...
};

MyClass<Q> x;      // OK
```

In this case, the `ptr` member of `MyClass<Q>` would be a pointer to type `int`. However, the subtype could also be an abstract data type, such as a class:

```
class Q {
    class SubType;
    ...
};
```

Note that `typename` is always necessary to qualify an identifier of a template as being a type, even if an interpretation that is not a type would make no sense. Thus, the general rule in C++ is that any identifier of a template is considered to be a value except if it is qualified by `typename`.

Apart from this, `typename` can also be used instead of `class` in a template declaration:

```
template <typename T> class MyClass;
```

Member Templates

Member functions of classes may be templates. However, member templates may not be virtual. For example:

```
class MyClass {
    ...
    template <typename T>
    void f(T);
};
```

Here, `MyClass::f` declares a set of member functions for parameters of any type. You can pass any argument as long as its type provides all operations used by `f()`.

This feature is often used to support automatic type conversions for members in class templates. For example, in the following definition, the argument `x` of `assign()` must have exactly the same type as the object it is called for:

[Click here to view code image]

```
template <typename T>
class MyClass {
  private:
    T value;
  public:
    void assign (const MyClass<T>& x) {  // x must have same type as *this
        value = x.value;
    }
    ...
};
```

It would be an error to use different template types for the objects of the `assign()` operation even if an automatic type conversion from one type to the other is provided:

[Click here to view code image]

```
void f()
{
    MyClass<double> d;
    MyClass<int> i;

    d.assign(d);   // OK
    d.assign(i);   // ERROR: i is MyClass<int>
                   //        but MyClass<double> is required
}
```

By providing a different template type for the member function, you relax the rule of exact match. The member function template argument may have any template type, then, as long as the types are assignable:

[Click here to view code image]

```
template <typename T>
class MyClass {
  private:
    T value;
  public:
    template <typename X>                // member template
    void assign (const MyClass<X>& x) {  // allows different template types
        value = x.getValue();
    }
    T getValue () const {
        return value;
    }
    ...
};
void f()
{
    MyClass<double> d;
    MyClass<int> i;

    d.assign(d);   // OK
    d.assign(i);   // OK (int is assignable to double)
```

```
}
```

Note that the argument `x` of `assign()` now differs from the type of `*this`. Thus, you can't access private and protected members of `MyClass<>` directly. Instead, you have to use something like `getValue()` in this example.

A special form of a member template is a *template constructor*. Template constructors are usually provided to enable implicit type conversions when objects are copied. Note that a template constructor does not suppress the implicit declaration of the copy constructor. If the type matches exactly, the implicit copy constructor is generated and called. For example:

Click here to view code image

```
template <typename T>
class MyClass {
  public:
    // copy constructor with implicit type conversion
    // - does not suppress implicit copy constructor
    template <typename U>
    MyClass (const MyClass<U>& x);
    ...
};

void f()
{
    MyClass<double> xd;
    ...
    MyClass<double> xd2(xd);    // calls implicitly generated copy constructor
    MyClass<int> xi(xd);        // calls template constructor
    ...
}
```

Here, the type of `xd2` is the same as the type of `xd` and so is initialized via the implicitly generated copy constructor. The type of `xi` differs from the type of `xd` and so is initialized by using the template constructor. Thus, if you implement a template constructor, don't forget to provide a default constructor if its default behavior does not fit your needs. See Section 5.1.1, page 60, for another example of member templates.

Nested Class Templates

Nested classes may also be templates:

```
template <typename T>
class MyClass {
    ...
    template <typename T2>
    class NestedClass;
    ...
};
```

3.2.1. Explicit Initialization for Fundamental Types

If you use the syntax of an explicit constructor call without arguments, fundamental types are initialized with zero:

```
int i1;            // undefined value
int i2 = int();    // initialized with zero
int i3{};          // initialized with zero (since C++11)
```

This feature enables you to write template code that ensures that values of any type have a certain default value. For example, in the following function, the initialization guarantees that `x` is initialized with zero for fundamental types:

```
template <typename T>
void f()
{
     T x = T();
     ...
}
```

If a template forces the initialization with zero, its value is so-called *zero initialized*. Otherwise it's *default initialized*.

3.2.2. Definition of `main()`

I'd also like to clarify an important, often misunderstood, aspect of the core language: namely, the only correct and portable versions of `main()`. According to the C++ standard, only two definitions of `main()` are portable:

```
int main()
{
    ...
```

```
}
```

and

```
int main (int argc, char* argv[])
{
    ...
}
```

where `argv` (the array of command-line arguments) might also be defined as `char**`. Note that the return type `int` is required.

You may, but are not required to, end `main()` with a `return` statement. Unlike C, C++ defines an implicit

```
return 0;
```

at the end of `main()`. This means that every program that leaves `main()` without a `return` statement is successful. Any value other than `0` represents a kind of failure (see Section 5.8.2, page 162, for predefined values). Therefore, my examples in this book have no `return` statement at the end of `main()`.

To end a C++ program without returning from `main()`, you usually should call `exit()`, `quick_exit()` (since C++11), or `terminate()`. See Section 5.8.2, page 162, for details.