
3.1. New C++11 Language Features

3.1.1. Important Minor Syntax Cleanups

First, I'd like to introduce two new features of C++11 that are minor but important for your day-today programming.

Spaces in Template Expressions

The requirement to put a space between two closing template expressions has gone:

```
vector<list<int> >;     // OK in each C++ version
vector<list<int>>;      // OK since C++11
```

Throughout the book (as in real code) you will find both forms.

`nullptr` and `std::nullptr_t`

C++11 lets you use `nullptr` instead of `0` or `NULL` to specify that a pointer refers to no value (which differs from having an undefined value). This new feature especially helps to avoid mistakes that occurred when a null pointer was interpreted as an integral value. For example:

Click here to view code image

```
void f(int);
void f(void*);

f(0);           // calls f(int)
f(NULL);        // calls f(int) if NULL is 0, ambiguous otherwise
f(nullptr);     // calls f(void*)
```

`nullptr` is a new keyword. It automatically converts into each pointer type but not to any integral type. It has type `std::nullptr_t`, definedin `<cstddef>` (see Section 5.8.1, page 161), so you can now even overload operations for the case that a null pointer is passed. Note that `std::nullptr_t` counts as a fundamental data type (see Section 5.4.2, page 127).

3.1.2. Automatic Type Deduction with `auto`

With C++11, you can declare a variable or an object without specifying its specific type by using `auto`.[1] For example:

```
auto i = 42;      // i has type int
double f();
auto d = f();     // d has type double
```

[1] Note that `auto` is an old keyword of C. As the counterpart of `static`, declaring a variable as local, it was never used, because not specifying something as static implicitly declared it as `auto`.

The type of a variable declared with `auto` is deduced from its initializer. Thus, an initialization is required:

```
auto i;           // ERROR: can't deduce the type of i
```

Additional qualifiers are allowed. For example:

```
static auto vat = 0.19;
```

Using `auto` is especially useful where the type is a pretty long and/or complicated expression. For example:

Click here to view code image

```
vector<string> v;
...
auto pos = v.begin();     // pos has type vector<string>::iterator

auto l = [] (int x) -> bool {    // l has the type of a lambda
             ...,                // taking an int and returning a bool
         };
```

The latter is an object, representing a lambda, which is introduced in Section 3.1.10, page 28.

3.1.3. Uniform Initialization and Initializer Lists

Before C++11, programmers, especially novices, could easily become confused by the question of how to initialize a variable or an object. Initialization could happen with parentheses, braces, and/or assignment operators.

For this reason, C++11 introduced the concept of uniform initialization, which means that for any initialization, you can use one common syntax. This syntax uses braces, so the following is possible now:

Click here to view code image

```
int values[] { 1, 2, 3 };
std::vector<int> v { 2, 3, 5, 7, 11, 13, 17 };
std::vector<std::string> cities {
    "Berlin", "New York", "London", "Braunschweig", "Cairo", "Cologne"
};
std::complex<double> c{4.0,3.0};  // equivalent to c(4.0,3.0)
```

An initializer list forces so-called *value initialization*, which means that even local variables of fundamental data types, which usually have an undefined initial value, are initialized by zero (or `nullptr`, if it is a pointer):

```
int i;          // i has undefined value
int j{};        // j is initialized by 0
int* p;         // p has undefined value
int* q{};       // q is initialized by nullptr
```

Note, however, that *narrowing* initializations — those that reduce precision or where the supplied value gets modified — are not possible with braces. For example:

Click here to view code image

```
int x1(5.3);            // OK, but OUCH: x1 becomes 5
int x2 = 5.3;           // OK, but OUCH: x2 becomes 5
int x3{5.0};            // ERROR: narrowing
int x4 = {5.3};         // ERROR: narrowing
char c1{7};             // OK: even though 7 is an int, this is not narrowing
char c2{99999};         // ERROR: narrowing (if 99999 doesn't fit into a char)
std::vector<int> v1 { 1, 2, 4, 5 };         // OK
std::vector<int> v2 { 1, 2.3, 4, 5.6 };     // ERROR: narrowing doubles to
ints
```

As you can see, to check whether narrowing applies, even the current values might be considered, if available at compile time. As Bjarne Stroustrup writes in [*Stroustrup:FAQ*] regarding this example: "The way C++11 avoids a lot of incompatibilities is by relying on the actual values of initializers (such as `7` in the example above) when it can (and not just type) when deciding what is a narrowing conversion. If a value can be represented exactly as the target type, the conversion is not narrowing. Note that floating-point to integer conversions are always considered narrowing — even `7.0` to `7`."

To support the concept of initializer lists for user-defined types, C++11 provides the class template `std::initializer_list<>`. It can be used to support initializations by a list of values or in any other place where you want to process just a list of values. For example:

Click here to view code image

```
void print (std::initializer_list<int> vals)
{
    for (auto p=vals.begin(); p!=vals.end(); ++p) {  // process a list of
values
        std::cout << *p << "\n";
    }
}

print ({12,3,5,7,11,13,17});  // pass a list of values to print()
```

When there are constructors for both a specific number of arguments and an initializer list, the version with the initializer list is preferred:

```
class P
{
  public:
    P(int,int);
    P(std::initializer_list<int>);
};

P p(77,5);          // calls P::P(int,int)
P q{77,5};          // calls P::P(initializer_list)
P r{77,5,42};       // calls P::P(initializer_list)
P s = {77,5};       // calls P::P(initializer_list)
```

Without the constructor for the initializer list, the constructor taking two `int`s would be called to initialize `q` and `s`, while the initialization of `r` would be invalid.

Because of initializer lists, `explicit` now also becomes relevant for constructors taking more than one argument. So, you can now disable automatic type conversions from multiple values, which is also used when an initialization uses the `=` syntax:

Click here to view code image

```
class P
{
  public:
    P(int a, int b) {
        ...
    }
    explicit P(int a, int b, int c) {
        ...
    }
};

P x(77,5);          // OK
P y{77,5};          // OK
P z {77,5,42};      // OK
P v = {77,5};       // OK (implicit type conversion allowed)
P w = {77,5,42};    // ERROR due to explicit (no implicit type conversion allowed)

void fp(const P&);

fp({47,11});        // OK, implicit conversion of {47,11} into P
fp({47,11,3});      // ERROR due to explicit
fp(P{47,11});       // OK, explicit conversion of {47,11} into P
fp(P{47,11,3});     // OK, explicit conversion of {47,11,3} into P
```

In the same manner, an `explicit` constructor taking an initializer list disables implicit conversions for initializer lists with zero, one, or more initial values.

3.1.4. Range-Based `for` Loops

C++11 introduces a new form of `for` loop, which iterates over all elements of a given range, array, or collection. It's what in other programming languages would be called a *foreach* loop. The general syntax is as follows:

```
for ( decl : coll ) {
    statement
}
```

where *decl* is the declaration of each element of the passed collection *coll* and for which the statements specified are called. For example, the following calls for each value of the passed initializer list the specified statement, which writes it on a line to the standard output `cout`:

```
for ( int i : { 2, 3, 5, 7, 9, 13, 17, 19 } ) {
    std::cout << i << std::endl;
}
```

To multiply each element `elem` of a vector `vec` by `3` you can program as follows:

```
std::vector<double> vec;
...
for ( auto& elem : vec ) {
    elem *= 3;
}
```

Here, declaring `elem` as a reference is important because otherwise the statements in the body of the `for` loop act on a local copy of the elements in the vector (which sometimes also might be useful).

This means that to avoid calling the copy constructor and the destructor for each element, you should usually declare the current element to be a constant reference. Thus, a generic function to print all elements of a collection should be implemented as follows:

```
template <typename T>
void printElements (const T& coll)
{
    for (const auto& elem : coll) {
        std::cout << elem << std::endl;
    }
}
```

Here, the range-based `for` statement is equivalent to the following:

Click here to view code image

```
{
    for (auto _pos=coll.begin(); _pos != coll.end(); ++_pos ) {
        const auto& elem = *_pos;
        std::cout << elem << std::endl;
    }
}
```

In general, a range-based `for` loop declared as

```
for ( decl : coll ) {
    statement
}
```

is equivalent to the following, if *coll* provides `begin()` and `end()` members:

Click here to view code image

```
{
     for (auto _pos=coll.begin(), _end=coll.end(); _pos!=_end; ++_pos ) {
         decl = *_pos;
         statement
     }
}
```

or, if that doesn't match, to the following by using a global `begin()` and `end()` taking *coll* as argument:

Click here to view code image

```
{
   for (auto _pos=begin(coll), _end=end(coll); _pos!=_end; ++_pos ) {
       decl = *_pos;
       statement
   }
}
```

As a result, you can use range-based `for` loops even for initializer lists because the class template `std::initializer_list<>` provides `begin()` and `end()` members.

In addition, there is a rule that allows you to use ordinary C-style arrays of known size. For example:

Click here to view code image

```
int array[] = { 1, 2, 3, 4, 5 };

long sum=0;     // process sum of all elements
for (int x : array) {
    sum += x;
}

for (auto elem : { sum, sum*2, sum*4 } ) {   // print some multiples of sum
    std::cout << elem << std::endl;
}
```

has the following output:

```
15
30
60
```

Note that no explicit type conversions are possible when elements are initialized as *decl* inside the `for` loop. Thus, the following does not compile:

Click here to view code image

```
class C
{
  public:
    explicit C(const std::string& s);  // explicit(!) type conversion from strings
    ...
};

std::vector<std::string> vs;
for (const C& elem : vs) {     // ERROR, no conversion from string to C defined
    std::cout << elem << std::endl;
}
```

3.1.5. Move Semantics and Rvalue References

One of the most important new features of C++11 is the support of move semantics. This feature goes further into the major design goal of C++ to avoid unnecessary copies and temporaries.

This new feature is so complex that I recommend using a more detailed introduction to this topic, but I will try to give a brief introduction and summary here.[2]

[2] This introduction is based on [*Abrahams:RValues*] (with friendly permission by Dave Abrahams), on [*Becker:RValues*] (with friendly permission by Thomas Becker), and some emails exchanged with Daniel Krügler, Dietmar Kühl, and Jens Maurer.

Consider the following code example:

```
void createAndInsert (std::multiset<X>& coll)
{
    X x;   // create an object of type X
    ...
    coll.insert(x); // insert it into the passed collection
}
```

Here, we insert a new object into a collection, which provides a member function that creates an internal copy of the passed element:

```
namespace std {
    template <typename T, ...> class multiset {
      public:
        ... insert (const T& v);   // copy value of v
        ...
    };
}
```

This behavior is useful because the collection provides value semantics and the ability to insert temporary objects or objects that are used and modified after being inserted:

Click here to view code image

```
X x;
coll.insert(x);      // inserts copy of x
...
coll.insert(x+x);    // inserts copy of temporary rvalue
...
coll.insert(x);      // inserts copy of x (although x is not used any longer)
```

However, for the last two insertions of `x`, it would be great to specify a behavior that the passed values (the result of `x+x` and `x`) are no longer used by the caller so that `coll` internally could avoid creating a copy and somehow *move* the contents of them into its new elements. Especially when copying `x` is expensive — for example, if it is a large collection of strings — this could become a big performance improvement.

Since C++11, such a behavior is possible. The programmer, however, has to specify that a move is possible unless a temporary is used. Although a compiler might find this out in trivial cases, allowing the programmer to perform this task lets this feature be used in all cases, where logically appropriate. The preceding code simply has to get modified as follows:

Click here to view code image

```
X x;
coll.insert(x);      // inserts copy of x (OK, x is still used)
...
coll.insert(x+x);    // moves (or copies) contents of temporary rvalue
...
coll.insert(std::move(x));   // moves (or copies) contents of x into coll
```

With `std::move()`, declared in `<utility>`, `x` can be *moved* instead of being copied. However, `std::move()` doesn't itself do any moving, but merely converts its argument into a so-called *rvalue reference*, which is a type declared with two ampersands: `X&&`. This new type stands for rvalues (anonymous temporaries that can appear only on the right-hand side of an assignment) that can be modified. The contract is that this is a (temporary) object that is not needed any longer so that you can *steal* its contents and/or its resources.

Now, the collection can provide an overloaded version of `insert()`, which deals with these rvalue references:

Click here to view code image

```
namespace std {
  template <typename T, ...> class multiset {
    public:
      ... insert (const T& x);   // for lvalues: copies the value
      ... insert (T&& x);        // for rvalues: moves the value
      ...
  };
```

```
}
```

The version for rvalue references can now be optimized so that its implementation *steals* the contents of `X`. To do that, however, we need the help of the type of `X`, because only the type of `X` has access to its internals. So, for example, you could use internal arrays and pointers of `X` to initialize the inserted element, which would be a huge performance improvement if class `X` is itself a complex type, where you had to copy element-by-element instead. To initialize the new internal element, we simply call a so-called *move constructor* of class `X`, which *steals* the value of the passed argument to initialize a new object. All complex types should — and in the C++ standard library will — provide such a special constructor, which moves the contents of an existing element to a new element:

```
class X {
  public:
    X (const X& lvalue); // copy constructor
    X (X&& rvalue);      // move constructor
    ...
};
```

For example, the move constructor for strings typically just assigns the existing internal character array to the new object instead of creating a new array and copying all elements. The same applies to all collection classes: Instead of creating a copy of all elements, you just assign the internal memory to the new object. If no move constructor is provided, the copy constructor will be used.

In addition, you have to ensure that any modification — especially a destruction — of the passed object, where the value was *stolen* from, doesn't impact the state of the new object that now owns the value. Thus, you usually have to clear the contents of the passed argument (for example, by assigning `nullptr` to its internal member referring to its elements).

Clearing the contents of an object for which move semantics were called is, strictly speaking, not required, but not doing so makes the whole mechanism almost useless. In fact, for the classes of the C++ standard library in general, it is guaranteed that after a move, the objects are in a *valid but unspecified* state. That is, you can assign new values afterward, but the current value is not defined. For STL containers, it is guaranteed that containers where the value was moved from are empty afterward.

In the same way, any nontrivial class should provide both a copy assignment and a move assignment operator:

Click here to view code image

```
class X {
  public:
    X& operator= (const X& lvalue);  // copy assignment operator
    X& operator= (X&& rvalue);       // move assignment operator
    ...
};
```

For strings and collections these operators could be implemented by simply swapping the internal contents and resources. However, you should also clear the contents of `*this` because this object might hold resources, such as locks, for which it is better to release them sooner. Again, the move semantics don't require that, but it is a quality of move support that, for example, is provided by the container classes of the C++ standard library.

Finally, note the following two remarks about this feature: (1) overloading rules for rvalue and lvalue references and (2) returning rvalue references.

Overloading Rules for Rvalue and Lvalue References

The overloading rules for rvalues and lvalues are as follows:[3]

[3] Thanks to Thomas Becker for providing this wording.

- If you implement only

  ```
  void foo(X&);
  ```

  without `void foo(X&&)`, the behavior is as in C++98: `foo()` can be called for lvalues but not for rvalues.

- If you implement

  ```
  void foo(const X&);
  ```

  without `void foo(X&&)`, the behavior is as in C++98: `foo()` can be called for rvalues and for lvalues.

- If you implement

  ```
  void foo(X&);
  void foo(X&&);
  ```

  or

  ```
  void foo(const X&);
  void foo(X&&);
  ```

  you can distinguish between dealing with rvalues and lvalues. The version for rvalues is allowed to and should provide move semantics. Thus, it can *steal* the internal state and resources of the passed argument.

- If you implement

```
void foo(X&&);
```

but neither `void foo(X&)` nor `void foo(const X&)`, `foo()` can be called on rvalues, but trying to call it on an lvalue will trigger a compile error. Thus, only move semantics are provided here. This ability is used inside the library: for example, by unique pointers (see Section 5.2.5, page 98), file streams (see Section 15.9.2, page 795), or string streams (see Section 15.10.2, page 806),

This means that if a class does not provide move semantics and has only the usual copy constructor and copy assignment operator, these will be called for rvalue references. Thus, `std::move()` means to call move semantics, if provided, and copy semantics otherwise.

Returning Rvalue References

You don't have to and should not `move()` return values. According to the language rules, the standard specifies that for the following code:[4]

```
X foo ()
{
    X x;
    ...
    return x;
}
```

[4] Thanks to Dave Abrahams for providing this wording.

the following behavior is guaranteed:

- If `X` has an accessible copy or move constructor, the compiler may choose to elide the copy. This is the so-called *(named) return value optimization* (*(N)RVO*), which was specified even before C++11 and is supported by most compilers.
- Otherwise, if `X` has a move constructor, `x` is moved.
- Otherwise, if `X` has a copy constructor, `x` is copied.
- Otherwise, a compile-time error is emitted.

Note also that returning an rvalue reference is an error if the returned object is a local nonstatic object:

```
X&& foo ()
{
    X x;
    ...
    return x;      // ERROR: returns reference to nonexisting object
}
```

An rvalue reference is a reference, and returning it while referring to a local object means that you return a reference to an object that doesn't exist any more. Whether `std::move()` is used doesn't matter.

3.1.6. New String Literals

Since C++11, you can define raw string and multibyte/wide-character string literals.

Raw String Literals

Such a raw string allows one to define a character sequence by writing exactly its contents as a raw character sequence. Thus, you save a lot of escapes necessary to mask special characters.

A raw string starts with `R"(` and ends with `)"`. The string might contain line breaks. For example, an ordinary string literal representing two backslashes and an `n` would be defined as an ordinary string literal as follows:

```
"\\\\n"
```

and as a raw string literal as follows:

```
R"(\\n)"
```

To be able to have `)"` inside the raw string, you can use a delimiter. Thus, the complete syntax of a raw string is `R"` *delim* `(` ... `)` *delim* `"`, where *delim* is a character sequence of at most 16 basic characters except the backslash, whitespaces, and parentheses.

For example, the raw string literal

```
R"nc(a\
     b\nc()"
     )nc";
```

is equivalent to the following ordinary string literal:

```
"a\\\n    b\\nc()\"\n    "
```

Thus, the string contains an `a`, a backslash, a newline character, some spaces, a `b`, a backslash, an `n`, a `c`, a double quote character, a newline character, and some spaces.

Raw string literals are especially useful when defining regular expressions. See Chapter 14 for details.

Encoded String Literals

By using an *encoding prefix*, you can define a special character encoding for string literals. The following encoding prefixes are defined:

- **u8** defines a UTF-8 encoding. A UTF-8 string literal is initialized with the given characters as encoded in UTF-8. The characters have type `const char`.
- **u** defines a string literal with characters of type `char16_t`.
- **U** defines a string literal with characters of type `char32_t`.
- **L** defines a wide string literal with characters of type `wchar_t`.

For example:

```
L"hello" // defines "hello" as wchar_t string literal
```

The initial `R` of a raw string can be preceded by an encoding prefix.

See Chapter 16 for details about using different encodings for internationalization.

3.1.7. Keyword `noexcept`

C++11 provides the keyword `noexcept`. It can be used to specify that a function cannot throw — or is not prepared to throw. For example:

```
void foo () noexcept;
```

declares that `foo()` won't throw. If an exception is not handled locally inside `foo()` — thus, if `foo()` throws — the program is terminated, calling `std::terminate()`, which by default calls `std::abort()` (see Section 5.8.2, page 162).

`noexcept` targets a lot of problems (empty) exception specifications have. To quote from [*N3051:DeprExcSpec*] (with friendly permission by Doug Gregor):

- *Runtime checking*: C++ exception specifications are checked at runtime rather than at compile time, so they offer no programmer guarantees that all exceptions have been handled. The runtime failure mode (calling `std::unexpected()`) does not lend itself to recovery.
- *Runtime overhead*: Runtime checking requires the compiler to produce additional code that also hampers optimizations.
- *Unusable in generic code*: Within generic code, it is not generally possible to know what types of exceptions may be thrown from operations on template arguments, so a precise exception specification cannot be written.

In practice, only two forms of exception-throwing guarantees are useful: An operation might throw an exception (any exception) or an operation will never throw any exception. The former is expressed by omitting the exception-specification entirely, while the latter can be expressed as `throw()` but rarely is, due to performance considerations.

Especially because `noexcept` does not require stack unwinding, programmers can now express the nothrow guarantee without additional overhead. As a result, the use of exception specifications is deprecated since C++11.

You can even specify a condition under which a function throws no exception. For example, for any type *Type*, the global `swap()` usually is defined as follows:

Click here to view code image

```
void swap (Type& x, Type& y) noexcept(noexcept(x.swap(y)))
{
    x.swap(y);
}
```

Here, inside `noexcept( ... )`, you can specify a Boolean condition under which no exception gets thrown: Specifying `noexcept` without condition is a short form of specifying `noexcept(true)`.

In this case, the condition is `noexcept(x.swap(y))`. Here, the *operator* `noexcept` is used, which yields `true` if an evaluated expression, which is specified within parentheses, can't throw an exception. Thus, the global `swap()` specifies that it does not throw an exception if the member function `swap()` called for the first argument does not throw.

As another example, the move assignment operator for value pairs is declared as follows:

Click here to view code image

```
pair& operator= (pair&& p)
        noexcept(is_nothrow_move_assignable<T1>::value &&
                 is_nothrow_move_assignable<T2>::value);
```

Here, the `is_nothrow_move_assignable` type trait is used, which checks whether for the passed type, a move assignment that does not throw is possible (see Section 5.4.2, page 127).

According to *[N3279:LibNoexcept]*, `noexcept` was introduced inside the library with the following conservative approach (words and phrases in *italics* are quoted literally):

- *Each library function ... that ... cannot throw* and does not specify any undefined behavior — for example, caused by a broken precondition — *should be marked as unconditionally* `noexcept` .
- *If a library swap function, move constructor, or move assignment operator ... can be proven not to throw by applying the* `noexcept` *operator, it should be marked as conditionally* `noexcept` *. No other function should use a conditional* `noexcept` *specification.*
- *No library destructor should throw. It must use the implicitly supplied (nonthrowing) exception specification.*
- *Library functions designed for compatibility with C code ... may be marked as unconditionally* `noexcept` .

Note that `noexcept` was deliberately not applied to any C++ function having a precondition that, if violated, could result in undefined behavior. This allows library implementations to provide a “safe mode” throwing a “precondition violation” exception in the event of misuse.

Throughout this book I usually skip `noexcept` specifications to improve readability and save space.

3.1.8. Keyword `constexpr`

Since C++11, `constexpr` can be used to enable that expressions be evaluated at compile time. For example:

```
constexpr int square (int x)
{
    return x * x;
}
float a[square(9)];  // OK since C++11: a has 81 elements
```

This keyword fixes a problem C++98 had when using numeric limits (see Section 5.3, page 115). Before C++11, an expression such as

```
std::numeric_limits<short>::max()
```

could not be used as an integral constant, although it was functionally equivalent to the macro `INT_MAX` . Now, with C++11, such an expression is declared as `constexpr` so that, for example, you can use it to declare arrays or in compile-time computations (metaprogramming):

```
std::array<float,std::numeric_limits<short>::max()> a;
```

Throughout this book I usually skip `constexpr` specifications to improve readability and save space.

3.1.9. New Template Features

Variadic Templates

Since C++11, templates can have parameters that accept a variable number of template arguments. This ability is called *variadic templates*. For example, you can use the following to call `print()` for a variable number of arguments of different types:

Click here to view code image

```
void print ()
{
}

template <typename T, typename... Types>
void print (const T& firstArg, const Types&... args)
{
    std::cout << firstArg << std::endl;  // print first argument
    print(args...);                      // call print() for remaining
arguments
}
```

If one or more arguments are passed, the function template is used, which by specifying the first argument separately allows the first argument to print and then recursively calls `print()` for the remaining arguments. To end the recursion, the non-template overload of `print()` is provided.

For example, an input such as

```
print (7.5, "hello", std::bitset<16>(377), 42);
```

would output the following (see Section 12.5.1, page 652 for details of bitsets):

```
7.5
hello
0000000101111001
42
```

Note that it is currently under discussion whether the following example also is valid. The reason is that formally for a single argument the variadic form is ambiguous with the nonvariadic form for a single argument; however, compilers usually accept this code:

Click here to view code image

```
template <typename T>
void print (const T& arg)
{
    std::cout << arg << std::endl;
}
template <typename T, typename... Types>
void print (const T& firstArg, const Types&... args)
{
    std::cout << firstArg << std::endl;  // print first argument
    print(args...);                      // call print() for remaining
arguments
}
```

Inside variadic templates, `sizeof...(args)` yields the number of arguments.

Class `std::tuple<>` makes heavy use of this feature (see Section 5.1.2, page 68).

Alias Templates (Template Typedef)

Since C++11, template (partial) type definitions also are supported. However, because all approaches with the `typename` keyword failed for some reason, the keyword `using` was introduced here, and the term *alias template* is used for it. For example, after

Click here to view code image

```
template <typename T>
using Vec = std::vector<T,MyAlloc<T>>;      // standard vector using own
allocator
```

the term

```
Vec<int> coll;
```

is equivalent to

```
std::vector<int,MyAlloc<int>> coll;
```

See Section 5.2.5, page 108, for another example.

Other New Template Features

Since C++11, function templates (see Section 3.2, page 34) may have default template arguments. In addition, local types can be used now as template arguments, and functions with internal linkage can now be used as arguments to nontype templates of function pointers or function references.

3.1.10. Lambdas

C++11 introduced *lambdas*, allowing the definition of inline functionality, which can be used as a parameter or a local object.

Lambdas change the way the C++ standard library is used. For example, Section 6.9, page 229, and Section 10.3, page 499, discuss how to use lambdas with STL algorithms and containers. Section 18.1.2, page 958, demonstrates how to use lambdas to define code that can run concurrently.

Syntax of Lambdas

A lambda is a definition of functionality that can be defined inside statements and expressions. Thus, you can use a lambda as an inline function.

The minimal lambda function has no parameters and simply does something. For example:

```
[] {
    std::cout << "hello lambda" << std::endl;
}
```

You can call it directly:

```
[] {
    std::cout << "hello lambda" << std::endl;
} ();          // prints "hello lambda"
```

or pass it to objects to get called:

```
auto l = [] {
             std::cout << "hello lambda" << std::endl;
         };
...
l();           // prints "hello lambda"
```

As you can see, a lambda is always introduced by a so-called *lambda introducer:* brackets within which you can specify a so-called *capture* to access nonstatic outside objects inside the lambda. When there is no need to have access to outside data, the brackets are just empty, as is the case here. Static objects such as `std::cout` can be used.

Between the lambda introducer and the lambda body, you can specify parameters, `mutable`, an exception specification, attribute specifiers, and the return type. All of them are optional, but if one of them occurs, the parentheses for the parameters are mandatory. Thus, the syntax of a lambda is either

```
[...]    {...}
```

or

```
[...]    (...) mutable_opt throwSpec_opt ->retType_opt {...}
```

A lambda can have parameters specified in parentheses, just like any other function:

```
auto l = [] (const std::string& s) {
             std::cout << s << std::endl;
         };
l("hello lambda");     // prints "hello lambda"
```

Note, however, that lambdas can't be templates. You always have to specify all types.

A lambda can also return something. Without any specific definition of the return type, it is deduced from the return value. For example, the return type of the following lambda is `int`:

```
[] {
    return 42;
}
```

To specify a return type, you can use the new syntax C++ also provides for ordinary functions (see Section 3.1.12, page 32). For example, the following lambda returns `42.0`:

```
[] () -> double {
    return 42;
}
```

In this case, you have to specify the return type after the parentheses for the arguments, which are required then, and the characters "`->`."

Between the parameters and the return specification or body, you can also specify an exception specification like you can do for functions. However, as for functions exception specifications are deprecated now (see Section 3.1.7, page 24).

Captures (Access to Outer Scope)

Inside the lambda introducer (brackets at the beginning of a lambda), you can specify a *capture* to access data of outer scope that is not passed as an argument:

- `[=]` means that the outer scope is passed to the lambda by value. Thus, you can read but not modify all data that was readable where the lambda was defined.
- `[&]` means that the outer scope is passed to the lambda by reference. Thus, you have write access to all data that was valid when the lambda was defined, provided that you had write access there.

You can also specify individually for each object that inside the lambda you have access to it by value or by reference. So, you can limit the access and mix different kinds of access. For example, the following statements:

```
int x=0;
int y=42;
auto qqq = [x, &y] {
               std::cout << "x: " << x << std::endl;
               std::cout << "y: " << y << std::endl;
               ++y;  // OK
           };
x = y = 77;
qqq();
```

```
qqq();
std::cout << "final y: " << y << std::endl;
```

produce the following output:

```
x: 0
y: 77
x: 0
y: 78
final y: 79
```

Because `x` gets passed by value, you are not allowed to modify it inside the lambda; calling `++x` inside the lambda would not compile. Because `y` is passed by reference, you have write access to it and are affected by any value change; so calling the lambda twice increments the assigned value `77`.

Instead of `[x, &y]`, you could also have specified `[=, &y]` to pass `y` by reference and all other objects by value.

To have a mixture of passing by value and passing by reference, you can declare the lambda as `mutable`. In that case, objects are passed by value, but inside the function object defined by the lambda, you have write access to the passed value. For example:

```
int id= 0;
auto f = [id] () mutable {
             std::cout << "id: " << id << std::endl;
             ++id;    // OK
         };
id= 42;
f();
f();
f();
std::cout << id << std::endl;
```

has the following output:

```
id: 0
id: 1
id: 2
42
```

You can consider the behavior of the lambda to be like the following function object (see Section 6.10, page 233):

```
class {
  private:
    int id;  // copy of outside id
  public:
    void operator() () {
            std::cout << "id: " << id << std::endl;
            ++id;    // OK
    }
};
```

Due to `mutable`, operator `()` is defined as a nonconstant member function, which means that write access to `id` is possible. So, with `mutable`, a lambda becomes stateful even if the state is passed by value. Without `mutable`, which is the usual case, operator `()` becomes a constant member function so that you only have read access to objects that were passed by value. See Section 10.3.2, page 501, for another example of using `mutable` with lambdas, which also discusses possible problems.

Type of Lambdas

The type of a lambda is an anonymous function object (or functor) that is unique for each lambda expression. Thus, to declare objects of that type, you need templates or `auto`. If you need the type, you can use `decltype()` (see Section 3.1.11, page 32), which is, for example, required to pass a lambda as hash function or ordering or sorting criterion to associative or unordered containers. See Section 6.9, page 232, and Section 7.9.7, page 379, for details.

Alternatively, you can use the `std::function<>` class template, provided by the C++ standard library, to specify a general type for functional programming (see Section 5.4.4, page 133). That class template provides the only way to specify the return type of a function returning a lambda:

```
// lang/lambda1.cpp

#include<functional>
#include<iostream>

std::function<int(int,int)> returnLambda ()
{
    return [] (int x, int y) {
               return x*y;
           };
```

```
}

int main()
{
    auto lf = returnLambda();
    std::cout << lf(6,7) << std::endl;
}
```

The output of the program is (of course):

```
42
```

3.1.11. Keyword `decltype`

By using the new `decltype` keyword, you can let the compiler find out the type of an expression. This is the realization of the often requested `typeof` feature. However, the existing `typeof` implementations were inconsistent and incomplete, so C++11 introduced a new keyword. For example:

```
std::map<std::string,float> coll;
decltype(coll)::value_type elem;
```

One application of `decltype` is to declare return types (see below). Another is to use it in metaprogramming (see Section 5.4.1, page 125) or to pass the type of a lambda (see Section 10.3.4, page 504).

3.1.12. New Function Declaration Syntax

Sometimes, the return type of a function depends on an expression processed with the arguments. However, something like

```
template <typename T1, typename T2>
decltype(x+y) add(T1 x, T2 y);
```

was not possible before C++11, because the return expression uses objects not introduced or in scope yet.

But with C++11, you can alternatively declare the return type of a function behind the parameter list:

```
template <typename T1, typename T2>
auto add(T1 x, T2 y) -> decltype(x+y);
```

This uses the same syntax as for lambdas to declare return types (see Section 3.1.10, page 28).

3.1.13. Scoped Enumerations

C++11 allows the definition of *scoped enumerations* — also called *strong enumerations*, or *enumeration classes* — which are a cleaner implementation of enumeration values (*enumerators*) in C++. For example:

```
enum class Salutation : char { mr, ms, co, none };
```

The important point is to specify keyword `class` behind `enum` .

Scoped enumerations have the following advantages:

- Implicit conversions to and from `int` are not possible.
- Values like `mr` are not part of the scope where the enumeration is declared. You have to use `Salutation::mr` instead.
- You can explicitly define the underlying type (`char` here) and have a guaranteed size (if you skip " `: char` " here, `int` is the default).
- Forward declarations of the enumeration type are possible, which eliminates the need to recompile compilation units for new enumerations values if only the type is used.

Note that with the type trait `std::underlying_type` , you can evaluate the underlying type of an enumeration type (see Section 5.4.2, page 130).

As an example, error condition values of standard exceptions are *scoped enumerators* (see Section 4.3.2, page 45).

3.1.14. New Fundamental Data Types

The following new fundamental data types are defined in C++11:

- `char16_t` and `char32_t` (see Section 16.1.3, page 852)
- `long long` and `unsigned long long`
- `std::nullptr_t` (see Section 3.1.1, page 14)