Programming Language

C++11

Summary of changes and general feature description.

Niedl, Markus
Markus.Niedl@gmx.net
2012-09-25

Contents

1 Preamble

This document is mainly a copy of the english version of Wikipedia article “C++11” and its subsequent articles “Variadic templates”, “Anonymous function” (chapter C++), “C++ Smart Pointer”, “Memory model (computing)”. The articles were gathered in August 2012. It was extended by some further examples and notes collected at conferences and trainings.

In first place it is thought to support a document that can easily be extended to the needs of a project or at least of a single developer. It could be extended by adding information about compiler, their versions and which features of C++11 are supported with it, or by adding project guidelines for each feature.

2 Abstract

C++11 (formerly known as C++0x[1]) is the most recent iteration of the C++ programming language. It was approved by ISO on 12 August 2011, replacing C++03.[2] The name is derived from the tradition of naming language versions by the year of the specification's publication.

C++11 includes several additions to the core language and extends the C++ standard library, incorporating most of the C++ Technical Report 1 (TR1) libraries — with the exception of the library of mathematical special functions.[3] C++11 was published as ISO/IEC 14882:2011[4] in September 2011 and is available for a fee. The working draft most similar to the published C++11 standard is N3337, dated 12 January 2012;[5] it has only editorial corrections from the C++11 standard.[6]

3 Changes from the previous version of the standard

The modifications for C++ involve both the core language and the standard library.

In the development of every utility of the 2011 standard, the committee has applied some directives:

- Maintain stability and compatibility with C++98 and possibly with C;
- Prefer introduction of new features through the standard library, rather than extending the core language;
- Prefer changes that can evolve programming technique;
- Improve C++ to facilitate systems and library design, rather than to introduce new features useful only to specific applications;
- Increase type safety by providing safer alternatives to earlier unsafe techniques;
- Increase performance and the ability to work directly with hardware;
- Provide proper solutions for real-world problems;

- Implement “zero-overhead” principle (additional support required by some utilities must be used only if the utility is used);
- Make C++ easy to teach and to learn without removing any utility needed by expert programmers.

Attention to beginners is considered important, because they will always compose the majority of computer programmers, and because many beginners would not intend to extend their knowledge of C++, limiting themselves to operate in the aspects of the language in which they are specialized.[1]

4 Extensions to the C++ core language

One function of the C++ committee is the development of the language core. Areas of the core language that were significantly improved include multithreading support, generic programming support, uniform initialization, and performance enhancements.

For the purposes of this article, core language features and changes are grouped into four general sections: run-time performance enhancements, build-time performance enhancements, usability enhancements, and new functionality. Some features could fall into multiple groups, but they are mentioned only in the group that primarily represents that feature.

5 Core language runtime performance enhancements

These language features primarily exist to provide some kind of performance benefit, either of memory or of computational speed.

5.1 Rvalue references and move constructors

In C++03 (and before), temporaries (termed "rvalues", as they often lie on the right side of an assignment) were intended to never be modifiable — just as in C — and were considered to be indistinguishable from const T& types; nevertheless, in some cases, temporaries could have been modified, a behavior that was even considered to be a useful loophole (for the former, see [7]). C++11 adds a new non-const reference type called an rvalue reference, identified by T&&. This refers to temporaries that are permitted to be modified after they are initialized, for the purpose of allowing "move semantics".

A chronic performance problem with C++03 is the costly and unnecessary deep copies that can happen implicitly when objects are passed by value. To illustrate the issue, consider that a std::vector<T> is, internally, a wrapper around a C-style array with a size. If a std::vector<T> temporary is created or returned from a function, it can be stored only by creating a new std::vector<T> and copying all of the rvalue's data into it. Then the temporary and all its memory is destroyed. (For simplicity, this discussion neglects the return value optimization).

In C++11, a "move constructor" of std::vector<T> that takes an rvalue reference to a std::vector<T> can copy the pointer to the internal C-style array out of the rvalue into the new std::vector<T>, then set the pointer inside the rvalue to null. Since the temporary will never again be used, no code will try to access the null pointer, and because the pointer is null, its memory is not deleted when it goes out of scope. Hence, the operation not only forgoes the expense of a deep copy, but is safe and invisible.

Rvalue references can provide performance benefits to existing code without needing to make any changes outside the standard library. The type of the returned value of a function returning a std::vector<T> temporary does not need to be changed explicitly to std::vector<T> && to invoke the move constructor, as temporaries are considered rvalues automatically. (However, if std::vector<T> is a C++03 version without a move constructor, then the copy constructor will be invoked with a const std::vector<T>& as normal, incurring a significant memory allocation.)

For safety reasons, some restrictions are imposed. A named variable will never be considered to be an rvalue even if it is declared as such; in order to get an rvalue, the function template std::move<T>() should be used. Rvalue references can also be modified only under certain circumstances, being intended to be used primarily with move constructors.

Due to the nature of the wording of rvalue references, and to some modification to the wording for lvalue references (regular references), rvalue references allow developers to provide perfect function forwarding. When combined with variadic templates, this ability allows for function templates that can perfectly forward arguments to another function that takes those particular arguments. This is most useful for forwarding constructor parameters, to create factory functions that will automatically call the correct constructor for those particular arguments. This is seen in the emplace_back set of STL methods.

5.2 constexpr - Generalized constant expressions

C++ has always had the concept of constant expressions. These are expressions such as 3+4 that will always yield the same results, at compile time and at run time. Constant expressions are optimization opportunities for compilers, and compilers frequently execute them at compile time and hardcode the results in the program. Also, there are a number of places where the C++ specification requires the use of constant expressions. Defining an array requires a constant expression, and enumerator values must be constant expressions.

However, a constant expression has never been allowed to contain a function call or object constructor. So a piece of code as simple as this is illegal:

```
int get_five() {return 5;}
// ### Create an array of 12 integers. Ill-formed C++03
int some_value[get_five() + 7];
```

This was not legal in C++03, because get_five() + 7 is not a constant expression. A C++03 compiler has no way of knowing if get_five() actually is constant at runtime. In theory, this function could affect a global variable, call other non-runtime constant functions, etc.

C++11 introduced the keyword constexpr, which allows the user to guarantee that a function or object constructor is a compile-time constant [8]. The above example can be rewritten as follows:

```
constexpr int get_five() {return 5;}
// ### Create an array of 12 integers. Legal C++11
int some_value[get_five() + 7];
```

This allows the compiler to understand, and verify, that get_five is a compile-time constant.

The use of constexpr on a function imposes some limitations on what that function can do. First, the function must have a non-void return type. Second, the function body cannot declare variables or define new types. Third, the body may contain only declarations, null statements and a single return statement. There must exist argument values such that, after argument substitution, the expression in the return statement produces a constant expression.

Prior to C++11, the values of variables could be used in constant expressions only if the variables are declared const, have an initializer which is a constant expression, and are of integral or enumeration type. C++11 removes the restriction that the variables must be of integral or enumeration type if they are defined with the constexpr keyword:

```
constexpr double earth_gravitational_acceleration = 9.8;
constexpr double moon_gravitational_acceleration =
   earth_gravitational_acceleration / 6.0;
```

Such data variables are implicitly const, and must have an initializer which must be a constant expression.

In order to construct constant expression data values from user-defined types, constructors can also be declared with constexpr. A constexpr constructor's function body can contain only declarations and null statements, and cannot declare variables or define types, as with a constexpr function. There must exist argument values such that, after argument substitution, it initializes the class's members with constant expressions. The destructors for such types must be trivial.

The copy constructor for a type with any constexpr constructors should usually also be defined as a constexpr constructor, in order to allow them to be returned by value from a constexpr function. Any member function of a class, such as copy constructors, operator overloads, etc., can be declared as constexpr, so long as they meet the requirements for constexpr functions. This allows the compiler to copy classes at compile time, perform operations on them, etc.

If a constexpr function or constructor is called with arguments which aren't constant expressions, the call behaves as if the function were not constexpr, and the resulting value is not a constant expression. Likewise, if the expression in the return statement of a constexpr function does not evaluate to a constant expression for a particular invocation, the result is not a constant expression.

5.3 Modification to the definition of plain old data

In C++03, a class or struct must follow a number of rules in order for it to be considered a plain old data (POD) type. Types that fit this definition produce object layouts that are compatible with C, and they could also be initialized statically. However, the definition in C++03 is unnecessarily strict and there are good reasons for allowing more types to fit the POD definition.

C++11 relaxed several of the POD rules, by dividing the POD concept into two separate concepts: trivial and standard-layout.

A type that is trivial can be statically initialized. It also means that it is legal to copy data around via memcpy, rather than having to use a copy constructor. The lifetime of a trivial type begins when its storage is defined, not when a constructor completes.

A trivial class or struct is defined as one that:

1. Has a trivial default constructor. This may use the default constructor syntax (SomeConstructor() = default;).
2. Has trivial copy and move constructors, which may use the default syntax.
3. Has trivial copy and move assignment operators, which may use the default syntax.
4. Has a trivial destructor, which must not be virtual.

Constructors are trivial only if there are no virtual member functions of the class and no virtual base classes. Copy/move operations also require that all of the non-static data members are trivial.

A type that is standard-layout means that it orders and packs its members in a way that is compatible with C. A class or struct is standard-layout, by definition, provided:

1. It has no virtual functions
2. It has no virtual base classes
3. All its non-static data members have the same access control (public, private, protected)
4. All its non-static data members, including any in its base classes, are in the same one class in the hierarchy

5. The above rules also apply to all the base classes and to all non-static data members in the class hierarchy
6. It has no base classes of the same type as the first defined non-static data member

A class/struct/union is considered POD if it is trivial, standard-layout, and all of its non-static data members and base classes are PODs.

By separating these concepts, it becomes possible to give up one without losing the other. A class with complex move and copy constructors may not be trivial, but it could be standard-layout and thus interop with C. Similarly, a class with public and private non-static data members would not be standard-layout, but it would be trivial and thus memcpy-able.

6 Core language build time performance enhancements

6.1 Extern template

In C++03, the compiler must instantiate a template whenever a fully specified template is encountered in a translation unit. If the template is instantiated with the same types in many translation units, this can dramatically increase compile times. There is no way to prevent this in C++03, so C++11 introduced extern template declarations, analogous to extern data declarations.

C++03 has this syntax to oblige the compiler to instantiate a template:

```
template class std::vector<MyClass>;
```

C++11 now provides this syntax:

```
extern template class std::vector<MyClass>;
```

which tells the compiler not to instantiate the template in this translation unit.

7 Core language usability enhancements

These features exist for the primary purpose of making the language easier to use. These can improve type safety, minimize code repetition, make erroneous code less likely, etc.

7.1 Initializer lists

C++03 inherited the initializer-list feature from C. A struct or array is given a list of arguments in curly brackets, in the order of the members' definitions in the struct. These initializer-lists are recursive, so an array of structs or struct containing other structs can use them.

```
struct Object
{
   float first;
   int second;
};
```

```
// ### One Object, with first=0.43f and second=10
Object scalar = {0.43f, 10};
// ### An array of three Objects
Object anArray[] = {{13.4f, 3}, {43.28f, 29}, {5.934f, 17}};
```

This is very useful for static lists or just for initializing a struct to a particular value. C++ also provides constructors to initialize an object, but they are often not as convenient as the initializer list. However C++03 allows initializer-lists only on structs and classes that conform to the Plain Old Data (POD) definition; C++11 extends initializer-lists, so they can be used for all classes including standard containers like std::vector.

C++11 binds the concept to a template, called std::initializer_list. This allows constructors and other functions to take initializer-lists as parameters. For example:

```
class SequenceClass
{
public:
   SequenceClass(std::initializer_list<int> list);
};
```

This allows SequenceClass to be constructed from a sequence of integers, as such:

```
SequenceClass some_var = {1, 4, 5, 6};
```

This constructor is a special kind of constructor, called an initializer-list-constructor. Classes with such a constructor are treated specially during uniform initialization (see below)

The class std::initializer_list<> is a first-class C++11 standard library type. However, they can be initially constructed statically by the C++11 compiler only through the use of the {} syntax. The list can be copied once constructed, though this is only a copy-by-reference. An initializer list is constant; its members cannot be changed once the initializer list is created, nor can the data in those members be changed.

Because initializer_list is a real type, it can be used in other places besides class constructors. Regular functions can take typed initializer lists as arguments. For example:

```
void function_name(std::initializer_list<float> list);

function_name({1.0f, -3.45f, -0.4f});
```

Standard containers can also be initialized in the following ways:

```
std::vector<std::string> v = { "xyzzy", "plugh", "abracadabra" };

std::vector<std::string> v({ "xyzzy", "plugh", "abracadabra" });

std::vector<std::string> v{ "xyzzy", "plugh", "abracadabra" };
// ### see "Uniform initialization" below
```

7.2 Uniform initialization

C++03 has a number of problems with initializing types. There are several ways to initialize types, and they do not all produce the same results when interchanged. The traditional constructor syntax, for example, can look like a function declaration, and steps must be taken to ensure that the compiler's most vexing parse rule will not mistake it for such. Only aggregates and POD types can be initialized with aggregate initializers (using SomeType var = {/*stuff*/};).

C++11 provides a syntax that allows for fully uniform type initialization that works on any object. It expands on the initializer list syntax:

```
struct BasicStruct
{
```

```
    int x;
    double y;
};

struct AltStruct
{
    AltStruct(int x, double y) : x_{x}, y_{y} {}

private:
    int x_;
    double y_;
};

BasicStruct var1{5, 3.2};
AltStruct var2{2, 4.3};
```

The initialization of var1 behaves exactly as though it were aggregate-initialization. That is, each data member of an object, in turn, will be copy-initialized with the corresponding value from the initializer-list. Implicit type conversion will be used where necessary. If no conversion exists, or only a narrowing conversion exists, the program is ill-formed. The initialization of var2 invokes the constructor.

One is also able to do the following:

```
struct IdString
{
    std::string name;
    int identifier;
};

IdString get_string()
{
    return {"foo", 42}; // ### Note the lack of explicit type.
}
```

Uniform initialization does not replace constructor syntax. There are still times when constructor syntax is required. If a class has an initializer list constructor (TypeName(initializer_list<SomeType>);), then it takes priority over other forms of construction, provided that the initializer list conforms to the sequence constructor's type. The C++11 version of std::vector has an initializer list constructor for its template type. This means that the following code:

```
std::vector<int> the_vec{4};
```

will call the initializer list constructor, not the constructor of std::vector that takes a single size parameter and creates the vector with that size. To access the latter constructor, the user will need to use the standard constructor syntax directly.

7.3 Type inference

In C++03 (and C), the type of a variable must be explicitly specified in order to use it. However, with the advent of template types and template metaprogramming techniques, the type of something, particularly the well-defined return value of a function, may not be easily expressed. Therefore, storing intermediates in variables is difficult, possibly requiring knowledge of the internals of a particular metaprogramming library.

C++11 allows this to be mitigated in two ways. First, the definition of a variable with an explicit initialization can use the auto keyword. This creates a variable of the specific type of the initializer:

```
auto some_strange_callable_type = boost::bind(&someFunction, _2, _1, some_object);
auto other_variable = 5;
```

The type of some_strange_callable_type is simply whatever the particular template function override of boost::bind returns for those particular arguments. This type is easily determined procedurally by the compiler as part of its semantic analysis duties, but is not easy for the user to determine upon inspection.

The type of other_variable is also well-defined, but it is easier for the user to determine. It is an int, which is the same type as the integer literal.

Additionally, the keyword decltype can be used to determine the type of an expression at compile-time. For example:

```
int some_int;
decltype(some_int) other_integer_variable = 5;
```

This is more useful in conjunction with auto, since the type of an auto variable is known only to the compiler. However, decltype can also be very useful for expressions in code that makes heavy use of operator overloading and specialized types.

auto is also useful for reducing the verbosity of the code. For instance, instead of writing

```
for(
   std::vector<int>::const_iterator itr = myvec.cbegin();
   itr != myvec.cend();
   ++itr )
```

the programmer can use the shorter

```
for( auto itr = myvec.cbegin(); itr != myvec.cend(); ++itr )
```

This difference grows as the programmer begins to nest containers, though in such cases typedefs are a good way to decrease the amount of code.

The type denoted by decltype can be different from the type deduced by auto.

```
#include <vector>
int main()
{
   const std::vector<int> v(1);

   // ### a has type int
   auto a = v[0];

   // ### b has type const int&, the return type of
   // ### std::vector<int>::operator[](size_type) const
   decltype(v[0]) b = 1;

   // ### c has type int
   auto c = 0;

   // ### d has type int
   auto d = c;

   // ### e has type int, the type of the
   // ###    entity named by c
   decltype(c) e;

   // ### f has type int&, because (c) is an lvalue
   decltype((c)) f = c;

   // ### g has type int, because 0 is an rvalue
   decltype(0) g;
}
```

7.4 Range-based for-loop

In C++03, iterating over the elements of a list requires a lot of code. Other languages have implemented support for syntactic sugar that allow the programmer to write a simple “foreach” statement that automatically traverses items in a list. One of those languages is the Java programming language, which received support for what has been defined as enhanced for loops in Java 5.0.[9]

C++11 added a similar feature. The statement for allows for easy iteration over a list of elements:

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

for( int &x : my_array )
{
   x *= 2;
}
```

This form of for, called the “range-based for”, will iterate over each element in the list. It will work for C-style arrays, initializer lists, and any type that has begin() and end() functions defined for it that return iterators. All of the standard library containers that have begin/end pairs will work with the range-based for statement.

7.5 Lambda functions and expressions

C++11 provides support for anonymous functions, called lambda functions in the specification.[10] A lambda expression has the form:

```
[capture](parameters)->return-type{body}
```

If there are no parameters the empty parentheses can be omitted. The return type can often be omitted, if the body consists only of one return statement or the return type is void.

```
[capture](parameters){body}
```

An example lambda function is defined as follows:

```
// ### implicit return type from 'return' statement
[](int x, int y) { return x + y; }

// ### no return statement -> lambda functions' return type is 'void'
[](int& x) { ++x; }

// ### no parameters, just accessing a global variable
[]() { ++global_x; }

// ### the same, so () can be omitted
[]{ ++global_x; }
```

The return type of this unnamed function is decltype(x+y). The return type can be omitted if the lambda function is of the form return expression (or if the lambda returns nothing), or if all locations that return a value return the same type when the return expression is passed through decltype.

The return type can be explicitly specified as follows:

```
[](int x, int y) -> int { int z = x + y; return z; }
```

In this example, a temporary variable, z, is created to store an intermediate. As with normal functions, the value of this intermediate is not held between invocations. Lambdas that return nothing can omit the return type specification; they do not need to use -> void.

A lambda function can refer to identifiers declared outside the lambda function. The set of these variables is commonly called a closure. Closures are defined between square brackets [and] in the declaration of lambda expression. The mechanism allows these variables to be captured by value or by reference. The following table demonstrates this:

`[]`	No variables defined. Attempting to use any external variables in the lambda is an error.
`[x, &y]`	Variable x is captured by value, y is captured by reference.
`[&]`	Any external variable is implicitly captured by reference if used.
`[=]`	Any external variable is implicitly captured by value if used.
`[&, x]`	Variable x is explicitly captured by value. Other variables will be captured by reference.
`[=, &z]`	Variable z is explicitly captured by reference. Other variables will be captured by value.

The following two examples demonstrate usage of a lambda expression:

```
std::vector<int> some_list;
int total = 0;
for(int i=0; i<5; ++i) some_list.push_back(i);
std::for_each(begin(some_list), end(some_list), [&total](int x)
{
  total += x;
});
```

This computes the total of all elements in the list. The variable total is stored as a part of the lambda function's closure. Since it is a reference to the stack variable total, it can change its value.

```
std::vector<int> some_list;
int total = 0;
int value = 5;
std::for_each(begin(some_list), end(some_list), [&, value, this](int x)
{
  total += x * value * this->some_func();
});
```

This will cause total to be stored as a reference, but value will be stored as a copy.

The capture of variable this is special. It can only be captured by value, not by reference. Therefore, when using the & specifier, this is not captured at all unless it is explicitly stated. Variable this can only be captured if the closest enclosing function is a non-static member function. The lambda will have the same access as the member that created it, in terms of protected/private members.

If variable this is captured, either explicitly or implicitly, then the scope of the enclosed class members is also tested. Accessing members of this does not require explicit use of this-> syntax.

The specific internal implementation can vary, but the expectation is that a lambda function that captures everything by reference will store the actual stack pointer of the function it is created in, rather than individual references to stack variables. However, because most lambda functions are small and local in scope, they are likely candidates for inlining, and thus will not need any additional storage for references.

If a closure object containing references to local variables is invoked after the innermost block scope of its creation, the behaviour is undefined.

Lambda functions are function objects of an implementation-dependent type; this type's name is only available to the compiler. If the user wishes to take a lambda function as a parameter, the type must be a template type, or they must create a std::function or a similar object to capture the lambda value. The use of the auto keyword can help store the lambda function,

```
auto my_lambda_func = [&](int x) { /*...*/ };
auto my_onheap_lambda_func = new auto([=](int x) { /*...*/ });
```

Here is an example of storing anonymous functions in variables, vectors, and arrays; and passing them as named parameters:

```
#include<functional>
#include<vector>
#include<iostream>

double eval(std::function<double(double)> f, double x = 2.0)
{
   return f(x);
}

int main()
{
   std::function<double(double)> f0 = [](double x){return 1;};
   auto                          f1 = [](double x){return x;};
   decltype(f0)               fa[3] = {f0,f1,[](double x){return x*x;}};
   std::vector<decltype(f0)>     fv = {f0,f1};
   fv.push_back([](double x){return x*x;});
   for(int i=0;i<fv.size();i++)  std::cout << fv[i](2.0) << "\n";
   for(int i=0;i<3;i++)          std::cout << fa[i](2.0) << "\n";
   for(auto &f : fv)             std::cout << f(2.0) << "\n";
   for(auto &f : fa)             std::cout << f(2.0) << "\n";
   std::cout << eval(f0) << "\n";
   std::cout << eval(f1) << "\n";
   return 0;
}
```

A lambda function with an empty capture specification ([]) can be implicitly converted into a function pointer with the same type as the lambda was declared with. So this is legal:

```
auto a_lambda_func = [](int x) { /*...*/ };
void(*func_ptr)(int) = a_lambda_func;
func_ptr(4); //calls the lambda.
```

7.6 Alternative function syntax

Standard C function declaration syntax was perfectly adequate for the feature set of the C language. As C++ evolved from C, it kept the basic syntax and extended it where necessary. However, as C++ became more complicated, it exposed a number of limitations, particularly with regard to template function declarations. The following, for example, is not allowed in C++03:

```
template<class Lhs, class Rhs>
   Ret adding_func(const Lhs &lhs, const Rhs &rhs) {return lhs + rhs;}
   // ### Ret must be the type of lhs+rhs
```

The type Ret is whatever the addition of types Lhs and Rhs will produce. Even with the aforementioned C++11 functionality of decltype, this is not possible:

```
template<class Lhs, class Rhs>
   decltype(lhs+rhs) adding_func(const Lhs &lhs, const Rhs &rhs)
   { return lhs + rhs; } // ### Not legal C++11
```

This is not legal C++ because lhs and rhs have not yet been defined; they will not be valid identifiers until after the parser has parsed the rest of the function prototype.

To work around this, C++11 introduced a new function declaration syntax, with a trailing-return-type:

```
template<class Lhs, class Rhs>
   auto adding_func(const Lhs &lhs, const Rhs &rhs) ->
      decltype(lhs+rhs) {return lhs + rhs;}
```

This syntax can be used for more mundane function declarations and definitions:

```
struct SomeStruct
{
   auto func_name(int x, int y) -> int;
};

auto SomeStruct::func_name(int x, int y) -> int
{
    return x + y;
}
```

The use of the keyword “auto” in this case means something different from its use in automatic type deduction.

7.7 Object construction improvement

In C++03, constructors of a class are not allowed to call other constructors of that class; each constructor must construct all of its class members itself or call a common member function, like these.

```
class SomeType
{
   int number;

public:
   SomeType(int new_number) : number(new_number) {}
   SomeType() : number(42) {}
};

class SomeType
{
    int number;

private:
   void Construct(int new_number) { number = new_number; }

public:
   SomeType(int new_number) { Construct(new_number); }
   SomeType() { Construct(42); }
};
```

Constructors for base classes cannot be directly exposed to derived classes; each derived class must implement constructors even if a base class constructor would be appropriate. Non-constant data members of classes cannot be initialized at the site of the declaration of those members. They can be initialized only in a constructor.

C++11 provides solutions to all of these problems.

C++11 allows constructors to call other peer constructors (known as delegation). This allows constructors to utilize another constructor's behavior with a minimum of added code. Examples of other languages similar to C++ that provide delegation are Java, C#, and D.

This syntax is as follows:

```
class SomeType
{
   int number;
```

```
public:
   SomeType(int new_number) : number(new_number) {}
   SomeType() : SomeType(42) {}
};
```

Notice that, in this case, the same effect could have been achieved by making new_number a defaulting parameter. The new syntax, however, allows the default value (42) to be expressed in the implementation rather than the interface — a benefit to maintainers of library code since default values for function parameters are “baked in” to call sites, whereas constructor delegation allows the value to be changed without recompilation of the code using the library.

This comes with a caveat: C++03 considers an object to be constructed when its constructor finishes executing, but C++11 considers an object constructed once any constructor finishes execution. Since multiple constructors will be allowed to execute, this will mean that each delegate constructor will be executing on a fully constructed object of its own type. Derived class constructors will execute after all delegation in their base classes is complete.

For base-class constructors, C++11 allows a class to specify that base class constructors will be inherited. This means that the C++11 compiler will generate code to perform the inheritance, the forwarding of the derived class to the base class. Note that this is an all-or-nothing feature; either all of that base class's constructors are forwarded or none of them are. Also, note that there are restrictions for multiple inheritance, such that class constructors cannot be inherited from two classes that use constructors with the same signature. Nor can a constructor in the derived class exist that matches a signature in the inherited base class.

The syntax is as follows:

```
class BaseClass
{
public:
   BaseClass(int value);
};

class DerivedClass : public BaseClass
{
public:
    using BaseClass::BaseClass;
};
```

For member initialization, C++11 allows the following syntax:

```
class SomeClass
{
public:
   SomeClass() {}
   explicit SomeClass(int new_value) : value(new_value) {}

private:
   int value = 5;
};
```

Any constructor of the class will initialize value with 5, if the constructor does not override the initialization with its own. So the above empty constructor will initialize value as the class definition states, but the constructor that takes an int will initialize it to the given parameter.

It can also use constructor or uniform initialization, instead of the equality initialization shown above.

7.8 Explicit overrides and final

In C++03, it is possible to accidentally create a new virtual function, when one intended to override a base class function. For example:

```
struct Base
{
   virtual void some_func(float);
};

struct Derived : Base
{
   virtual void some_func(int);
};
```

The Derived::some_func is intended to replace the base class version. But because it has a different interface, it creates a second virtual function. This is a common problem, particularly when a user goes to modify the base class.

C++11 provides syntax to solve this problem.

```
struct Base
{
   virtual void some_func(float);
};

struct Derived : Base
{
   // ### ill-formed because it doesn't override a base class method
   virtual void some_func(int) override;
};
```

The override special identifier means that the compiler will check the base class(es) to see if there is a virtual function with this exact signature. And if there is not, the compiler will error out.

C++11 also adds the ability to prevent inheriting from classes or simply preventing overriding methods in derived classes. This is done with the special identifier final. For example:

```
struct Base1 final { };

// ### ill-formed because the class Base1 has been marked final
struct Derived1 : Base1 { };

struct Base2
{
   virtual void f() final;
};

struct Derived2 : Base2
{
   // ### ill-formed because the
   // ### virtual function Base2::f has been marked final
   void f();
};
```

In this example, the virtual void f() final; statement declares a new virtual function, but it also prevents derived classes from overriding it. It also has the effect of preventing derived classes from using that particular function name and parameter combination.

Note that neither override nor final are language keywords. They are technically identifiers; they gain special meaning only when used in those specific contexts. In any other location, they can be valid identifiers.

7.9 Null pointer constant

For the purposes of this section and this section alone, every occurrence of “0” is meant as “a constant expression which evaluates to 0, which is of type int”. In reality, the constant expression can be of any integral type.

Since the dawn of C in 1972, the constant 0 has had the double role of constant integer and null pointer constant. The ambiguity inherent in the double meaning of 0 was dealt with in C by the use of the preprocessor macro NULL, which commonly expands to either ((void*)0) or 0. C++ didn't adopt the same behavior, allowing only 0 as a null pointer constant. This interacts poorly with function overloading:

```
void foo(char *);
void foo(int);
```

If NULL is defined as 0 (which is usually the case in C++), the statement foo(NULL); will call foo(int), which is almost certainly not what the programmer intended, and not what a superficial reading of the code suggests.

C++11 corrects this by introducing a new keyword to serve as a distinguished null pointer constant: nullptr. It is of type nullptr_t, which is implicitly convertible and comparable to any pointer type or pointer-to-member type. It is not implicitly convertible or comparable to integral types, except for bool. While the original proposal specified that an rvalue of type nullptr should not be convertible to bool, the core language working group decided that such a conversion would be desirable, for consistency with regular pointer types. The proposed wording changes were unanimously voted into the Working Paper in June 2008.[2]

For backwards compatibility reasons, 0 remains a valid null pointer constant.

```
char *pc = nullptr;     // ### OK
int  *pi = nullptr;     // ### OK
bool   b = nullptr;     // ### OK. b is false.
int    i = nullptr;     // ### error

foo(nullptr);           // ### calls foo(char *), not foo(int);
```

7.10 Strongly typed enumerations

In C++03, enumerations are not type-safe. They are effectively integers, even when the enumeration types are distinct. This allows the comparison between two enum values of different enumeration types. The only safety that C++03 provides is that an integer or a value of one enum type does not convert implicitly to another enum type. Additionally, the underlying integral type is implementation-defined; code that depends on the size of the enumeration is therefore non-portable. Lastly, enumeration values are scoped to the enclosing scope. Thus, it is not possible for two separate enumerations to have matching member names.

C++11 allows a special classification of enumeration that has none of these issues. This is expressed using the enum class (enum struct is also accepted as a synonym) declaration:

```
enum class Enumeration
{
    Val1,
    Val2,
    Val3 = 100,
    Val4 // = 101
};
```

This enumeration is type-safe. Enum class values are not implicitly converted to integers; therefore, they cannot be compared to integers either (the expression Enumeration::Val4 == 101 gives a compiler error).

The underlying type of enum classes is always known. The default type is int, this can be overridden to a different integral type as can be seen in the following example:

```
enum class Enum2 : unsigned int {Val1, Val2};
```

The scoping of the enumeration is also defined as the enumeration name's scope. Using the enumerator names requires explicitly scoping. Val1 is undefined, but Enum2::Val1 is defined.

Additionally, C++11 will allow old-style enumerations to provide explicit scoping as well as the definition of the underlying type:

```
enum Enum3 : unsigned long {Val1 = 1, Val2};
```

The enumerator names are defined in the enumeration's scope (Enum3::Val1).

Forward-declaring enums is also possible in C++11. Previously, enum types could not be forward-declared because the size of the enumeration depends on the definition of its members. As long as the size of the enumeration is specified either implicitly or explicitly, it can be forward-declared:

```
// ### Illegal in C++03 and C++11;
// ### the underlying type cannot be determined.
enum Enum1;

// ### Legal in C++11, the underlying type is explicitly specified.
enum Enum2 : unsigned int;

// ### Legal in C++11, the underlying type is int.
enum class Enum3;

// ### Legal in C++11.
enum class Enum4 : unsigned int;

// ### Illegal in C++11, because Enum2 was previously
// ### declared with a different underlying type.
enum Enum2 : unsigned short;
```

7.11 Right angle bracket

C++03's parser defines “>>” as the right shift operator in all cases. However, with nested template declarations, there is a tendency for the programmer to neglect to place a space between the two right angle brackets, thus causing a compiler syntax error.

C++11 improves the specification of the parser so that multiple right angle brackets will be interpreted as closing the template argument list where it is reasonable. This can be overridden by using parentheses:

```
template<bool Test> class SomeType;
// ### Interpreted as a std::vector of SomeType<true> 2>,
// ### which is not legal syntax. 1 is true.
std::vector<SomeType<1>2>> x1;

// ### Interpreted as std::vector of SomeType<false>,
// ### which is legal C++11 syntax. (1>2) is false.
std::vector<SomeType<(1>2)>> x1;
```

7.12 Explicit conversion operators

C++98 added the explicit keyword as a modifier on constructors to prevent single-argument constructors from being used as implicit type conversion operators. However, this does nothing for actual conversion operators. For example, a smart pointer class may have an operator bool() to allow it to act more like a primitive pointer: if it includes this conversion, it can be tested with if(smart_ptr_variable) (which would be true if the pointer was non-null and false otherwise). However, this allows other, unintended conversions as well. Because C++ bool is defined as an arithmetic type, it can be implicitly converted to integral or even floating-point types, which allows for mathematical operations that are not intended by the user.

In C++11, the explicit keyword can now be applied to conversion operators. As with constructors, it prevents the use of those conversion functions in implicit conversions. However, language contexts that specifically require a boolean value (the conditions of if-statements and loops, as well as operands to the logical operators) count as explicit conversions and can thus use a bool conversion operator.

7.13 Alias templates

In C++03, it is possible to define a typedef only as a synonym for another type, including a synonym for a template specialization with all actual template arguments specified. It is not possible to create a typedef template. For example:

```
template <typename First, typename Second, int Third>
class SomeType;

template <typename Second>
// ### Illegal in C++03
typedef SomeType<OtherType, Second, 5> TypedefName;
```

This will not compile.

C++11 adds this ability with the following syntax:

```
template <typename First, typename Second, int Third>
class SomeType;

template <typename Second>
using TypedefName = SomeType<OtherType, Second, 5>;
```

The using syntax can be also used as type aliasing in C++11:

```
typedef void (*Type)(double);           // ### Old style
using OtherType = void (*)(double);     // ### New introduced syntax
```

7.14 Unrestricted unions

In C++03, there are restrictions on what types of objects can be members of a union. For example, unions cannot contain any objects that define a non-trivial constructor. C++11 lifts some of these restrictions.[3]

This is a simple example of a union permitted in C++:

```
// ### for placement new
#include <new>

struct Point
{
   Point() {}
   Point(int x, int y): x_(x), y_(y) {}
```

```
    int x_, y_;
};

union U
{
    int z;
    double w;
    // ### Illegal in C++03,
    // ### point has a non-trivial constructor.
    // ### However, this is legal in C++11.
    Point p;
    // ### No nontrivial member functions
    // ### are implicitly defined for a union.
    // ### If required they are instead deleted
    // ### to force a manual definition.
    U() { new( &p ) Point(); }
};
```

The changes will not break any existing code since they only relax current rules.

8 Core language functionality improvements

These features allow the language to do things that were previously impossible, exceedingly verbose, or required non-portable libraries.

8.1 Variadic templates

Prior to C++11, templates (classes and functions) can only take a fixed number of arguments that have to be specified when a template is first declared. C++11 allows template definitions to take an arbitrary number of arguments of any type.

```
template<typename... Values> class tuple;
```

The above template class tuple will take any number of typenames as its template parameters. Here, an instance of the above template class is instantiated with three type arguments:

```
tuple<int, std::vector<int>,
    std::map<std::string, std::vector<int>>> some_instance_name;
```

The number of arguments can be zero, so tuple<> some_instance_name; will work as well.

If one does not want to have a variadic template that takes 0 arguments, then this definition will work as well:

```
template<typename First, typename... Rest> class tuple;
```

Variadic templates may also apply to functions, thus not only providing a type-safe add-on to variadic functions (such as printf) - but also allowing a printf-like function to process non-trivial objects.

```
template<typename... Params>
void printf(const std::string &str_format, Params... parameters);
```

The ... operator has two roles. When it occurs to the left of the name of a parameter, it declares a parameter pack. By using the parameter pack, user can bind zero or more arguments to the variadic template parameters. Parameter packs can also be used for non-type parameters. By contrast, when the ... operator occurs to the right of a template or function call argument, it unpacks the parameter packs into separate arguments, like the args... in the body of printf below. In practice, the use of ... operator in the code causes

that the whole expression that precedes the ... operator, will be repeated for every next argument unpacked from the argument pack, and all these expressions will be separated by a comma.

The use of variadic templates is often recursive. The variadic parameters themselves are not readily available to the implementation of a function or class. Therefore, the typical mechanism for defining something like a C++11 variadic printf replacement would be as follows:

```
void printf(const char *s)
{
   while (*s)
   {
      if (*s == '%' && *(++s) != '%')
      {
         throw std::runtime_error(
            "invalid format string: missing arguments");
      }
      std::cout << *s++;
   }
}

template<typename T, typename... Args>
void printf(const char *s, T value, Args... args)
{
   while (*s)
   {
      if (*s == '%' && *(++s) != '%')
      {
         std::cout << value;
         ++s;
         // ### call even when *s == 0 to detect extra arguments
         printf(s, args...);
         return;
      }
      std::cout << *s++;
   }
   throw std::logic_error("extra arguments provided to printf");
}
```

This is a recursive template. Notice that the variadic template version of printf calls itself, or (in the event that args... is empty) calls the base case.

There is no simple mechanism to iterate over the values of the variadic template. There are few methods to translate the argument pack into single argument use. Usually this will rely on function overloading, or - if your function can simply pick one argument at a time - using a dumb expansion marker:

```
template<typename... Args> inline void pass(Args&&...) {}
```

This way you can use it:

```
pass( someFunction(args)... );
```

which will expand to something like:

```
pass( someFunction(arg1), someFunction(arg2), someFunction(arg3), etc.);
```

The use of this "pass" function is necessary because the argument packs expands with separating by comma, but it can only be a comma of separating the function call arguments, not an "operator," function. Because of that "someFunction(args)...;" will never work. Moreover, this above solution will only work when someFunction return type isn't void and it will do all the someFunction calls in an unspecified order, because function argument evaluation order is not sequenced specifically. To avoid the unspecified order, brace

enclosed initializer lists can be used, which guarantee strict left to right order of evaluation. To avoid the need for a void return type, the comma operator can be used to always yield 1 in each expansion element.

```
struct pass
{
   template<typename ...T> pass(T...) {}
};

pass
{ (someFunction(args), 1)... };
```

Instead of executing a function, a lambda expression may be specified and executed in place, which allows executing arbitrary sequences of statements in-place. To date (01/2011), GCC does not support lambda expressions that contain unexpanded parameter packs yet though, so this cannot be used on that compiler yet.

```
pass{([&]{ std::cout << args << std::endl; }(), 1)...};
```

Another method is to use overloading with "termination versions" of functions. This method is more universal, but requires a bit more code and more effort to create. One function receives one argument of some type and the argument pack, the other does not have any of these two (if both have the same list of initial parameters, the call would be ambiguous - a variadic parameter pack alone cannot disambiguate a call):

```
int func() {} // ### termination version

template<typename Arg1, typename... Args>
int func(const Arg1& arg1, const Args&... args)
{
   process( arg1 );
   func(args...); // ### Note: arg1 does not appear here!
}
```

If args... contains at least one argument, it will redirect to the second version - parameter pack can be also empty, so if it's empty, it will simply redirect to the termination version, which will do nothing.

Variadic templates can be used also in exception specification, base class list and constructor's initialization list. For example, a class can specify the following:

```
template <typename... BaseClasses> class ClassName : public BaseClasses...
{
public:
   ClassName (BaseClasses&&... base_classes)
   : BaseClasses(base_classes)...
   {}
};
```

The unpack operator will replicate the types for the base classes of ClassName, such that this class will be derived from each of the types passed in. Also, the constructor must take a reference to each base class, so as to initialize the base classes of ClassName.

With regard to function templates, the variadic parameters can be forwarded. When combined with rvalue references (see above), this allows for perfect forwarding:

```
template<typename TypeToConstruct> struct SharedPtrAllocator
{
   template<typename ...Args>
   std::shared_ptr<TypeToConstruct> construct_with_shared_ptr(
      Args&&... params)
   {
      return std::shared_ptr<TypeToConstruct>(
         new TypeToConstruct(std::forward<Args>(params)...));
```

```
    };
};
```

This unpacks the argument list into the constructor of TypeToConstruct. The std::forward<Args>(params) syntax is the syntax that perfectly forwards arguments as their proper types, even with regard to rvalue-ness, to the constructor. The unpack operator will propagate the forwarding syntax to each parameter. This particular factory function automatically wraps the allocated memory in a std::shared_ptr for a degree of safety with regard to memory leaks.

Additionally, the number of arguments in a template parameter pack can be determined as follows:

```
template<typename ...Args> struct SomeStruct
{
    static const int size = sizeof...(Args);
};
```

The syntax SomeStruct<Type1, Type2>::size will be 2, while SomeStruct<>::size will be 0.

8.2 New string literals

C++03 offers two kinds of string literals. The first kind, contained within double quotes, produces a null-terminated array of type const char. The second kind, defined as L"", produces a null-terminated array of type const wchar_t, where wchar_t is a wide-character. Neither literal type offers support for string literals with UTF-8, UTF-16, or any other kind of Unicode encodings.

For the purpose of enhancing support for Unicode in C++ compilers, the definition of the type char has been modified to be both at least the size necessary to store an eight-bit coding of UTF-8 and large enough to contain any member of the compiler's basic execution character set. It was previously defined as only the latter.

There are three Unicode encodings that C++11 will support: UTF-8, UTF-16, and UTF-32. In addition to the previously noted changes to the definition of char, C++11 adds two new character types: char16_t and char32_t. These are designed to store UTF-16 and UTF-32 respectively.

The following shows how to create string literals for each of these encodings:

```
u8"I'm a UTF-8 string."
u"This is a UTF-16 string."
U"This is a UTF-32 string."
```

The type of the first string is the usual const char[]. The type of the second string is const char16_t[]. The type of the third string is const char32_t[].

When building Unicode string literals, it is often useful to insert Unicode codepoints directly into the string. To do this, C++11 allows the following syntax:

```
u8"This is a Unicode Character: \u2018."
u"This is a bigger Unicode Character: \u2018."
U"This is a Unicode Character: \u2018."
```

The number after the \u is a hexadecimal number; it does not need the usual 0x prefix. The identifier \u represents a 16-bit Unicode codepoint; to enter a 32-bit codepoint, use \U and a 32-bit hexadecimal number. Only valid Unicode codepoints can be entered. For example, codepoints on the range U+D800—U+DFFF are forbidden, as they are reserved for surrogate pairs in UTF-16 encodings.

It is also sometimes useful to avoid escaping strings manually, particularly for using literals of XML files, scripting languages, or regular expressions. C++11 provides a raw string literal:

```
R"(The String Data \ Stuff " )"
R"delimiter(The String Data \ Stuff " )delimiter"
```

In the first case, everything between the "(and the)" is part of the string. The " and \ characters do not need to be escaped. In the second case, the "delimiter(starts the string, and it ends only when)delimiter" is reached. The string delimiter can be any string up to 16 characters in length, including the empty string. This string cannot contain spaces, control characters, '(', ')', or the '\' character. The use of this delimiter string allows the user to have ")" characters within raw string literals. For example, R"delimiter((a-z))delimiter" is equivalent to "(a-z)".[4]

Raw string literals can be combined with the wide literal or any of the Unicode literal prefixes:

```
u8R"XXX(I'm a "raw UTF-8" string.)XXX"
uR"*(This is a "raw UTF-16" string.)*"
UR"(This is a "raw UTF-32" string.)"
```

8.3 User-defined literals

C++03 provides a number of literals. The characters “12.5” are a literal that is resolved by the compiler as a type double with the value of 12.5. However, the addition of the suffix “f”, as in “12.5f”, creates a value of type float that contains the value 12.5. The suffix modifiers for literals are fixed by the C++ specification, and C++ code cannot create new literal modifiers.

C++11 also includes the ability for the user to define new kinds of literal modifiers that will construct objects based on the string of characters that the literal modifies.

Literals transformation is redefined into two distinct phases: raw and cooked. A raw literal is a sequence of characters of some specific type, while the cooked literal is of a separate type. The C++ literal 1234, as a raw literal, is this sequence of characters '1', '2', '3', '4'. As a cooked literal, it is the integer 1234. The C++ literal 0xA in raw form is '0', 'x', 'A', while in cooked form it is the integer 10.

Literals can be extended in both raw and cooked forms, with the exception of string literals, which can be processed only in cooked form. This exception is due to the fact that strings have prefixes that affect the specific meaning and type of the characters in question.

All user-defined literals are suffixes; defining prefix literals is not possible.

User-defined literals processing the raw form of the literal are defined as follows:

```
OutputType operator "" _suffix(const char *literal_string);

OutputType some_variable = 1234_suffix;
```

The second statement executes the code defined by the user-defined literal function. This function is passed "1234" as a C-style string, so it has a null terminator.

An alternative mechanism for processing integer and floating point raw literals is through a variadic template:

```
template<char...> OutputType operator "" _suffix();

OutputType some_variable = 1234_suffix;
OutputType another_variable = 2.17_suffix;
```

This instantiates the literal processing function as operator "" _suffix<'1', '2', '3', '4'>(). In this form, there is no terminating null character to the string. The main purpose to doing this is to use C++11's constexpr keyword and the compiler to allow the literal to be transformed entirely at compile time, assuming OutputType is a constexpr-constructable and copyable type, and the literal processing function is a constexpr function.

For numeric literals, the type of the cooked literal is either unsigned long long for integral literals or long double for floating point literals. (Note: There is no need for signed integral types because a sign-prefixed literal is parsed as expression containing the sign as unary prefix operator and the unsigned number.) There is no alternative template form:

```
OutputType operator "" _suffix(unsigned long long);
OutputType operator "" _suffix(long double);

// ### uses the first function
OutputType some_variable    = 1234_suffix
// ### uses the second function
OutputType another_variable = 3.1416_suffix;
```

For string literals, the following are used, in accordance with the previously mentioned new string prefixes:

```
OutputType operator ""
   _suffix(const char * string_values, size_t num_chars);
OutputType operator ""
   _suffix(const wchar_t * string_values, size_t num_chars);
OutputType operator ""
   _suffix(const char16_t * string_values, size_t num_chars);
OutputType operator ""
   _suffix(const char32_t * string_values, size_t num_chars);

// ### Calls the const char * version
OutputType some_variable = "1234"_suffix;
// ### Calls the const char * version
OutputType some_variable = u8"1234"_suffix;
// ### Calls the const wchar_t * version
OutputType some_variable = L"1234"_suffix;
// ### Calls the const char16_t * version
OutputType some_variable = u"1234"_suffix;
// ### Calls the const char32_t * version
OutputType some_variable = U"1234"_suffix;
```

There is no alternative template form. Character literals are defined similarly.

8.4 Multithreading memory model

A memory model allows a compiler to perform many important optimizations. Even simple compiler optimizations like loop fusion move statements in the program can influence the order of read and write operations of potentially shared variables. Changes in the ordering of reads and writes can cause race conditions. Without a memory model, a compiler is not allowed to apply such optimizations to multi-threaded programs in general, or only in special cases.

Modern programming languages like Java therefore implement a memory model. The memory model specifies synchronization barriers that are established via special, well-defined synchronization operations such as acquiring a lock by entering a synchronized block or method. The memory model stipulates that changes to the values of shared variables only need to be made visible to other threads when such a synchronization barrier is reached. Moreover, the entire notion of a race condition is entirely defined over the order of operations with respect to these memory barriers.[17]

These semantics then give optimizing compilers a higher degree of freedom when applying optimizations: the compiler needs to make sure only that the values of (potentially shared) variables at synchronization barriers are guaranteed to be the same in both the optimized and unoptimized code. In particular, reordering statements in a block of code that contains no synchronization barrier is assumed to be safe by the compiler.

Most research in the area of memory models revolves around:

- Designing a memory model that allows a maximal degree of freedom for compiler optimizations while still giving sufficient guarantees about race-free and (perhaps more importantly) race-containing programs.

- Proving program optimizations that are correct with respect to such a memory model.

The Java Memory Model was the first attempt to provide a comprehensive threading memory model for a popular programming language.[18] Memory model semantics have since been standardized for the languages C++11 and C11, the current versions of C++ and C.[19][20]

In C++11 there are two parts involved: a memory model which allows multiple threads to co-exist in a program and library support for interaction between threads. (See this article's section on threading facilities.)

8.5 Thread-local storage

In a multi-threaded environment, it is common for every thread to have some unique variables. This already happens for the local variables of a function, but it does not happen for global and static variables.

A new thread-local storage duration (in addition to the existing static, dynamic and automatic) is indicated by the storage specifier thread_local.

Any object which could have static storage duration (i.e., lifetime spanning the entire execution of the program) may be given thread-local duration instead. The intent is that like any other static-duration variable, a thread-local object can be initialized using a constructor and destroyed using a destructor.

The specifier is allowed for namespace scope objects (e.g. global objects), file scope static objects, function local static objects, static data members of a class. Declaration of any other object as thread_local is an error.

The thread_local variables may be initialized either statically or dynamically:

```
thread_local int numA = 999;          // ### static initialization
thread_local std::string* ptrStr;     // ### static initialization
thread_local static char buf[200];    // ### static initialization

thread_local std::string str("Neo"); // ### dynamic initialization
thread_local int numB = func();       // ### dynamic initialization
```

Although it defines the lifetime and scope of an object it does not restrict the access to such a variable. This means a thread_local variable can be accessed from any other thread. But to get access to it the address of that variable has to be passed to another thread. It is guaranteed that the addresses are valid as long the corresponding thread is alive. As soon as the thread terminates, all thread_local addresses become invalid.

8.6 Explicitly defaulted and deleted special member functions

In C++03, the compiler provides, for classes that do not provide them for themselves, a default constructor, a copy constructor, a copy assignment operator (operator=), and a destructor. The programmer can override these defaults by defining custom versions. C++ also defines several global operators (such as operator= and operator new) that work on all classes, which the programmer can override.

However, there is very little control over the creation of these defaults. Making a class inherently non-copyable, for example, requires declaring a private copy constructor and copy assignment operator and not defining them. Attempting to use these functions is a violation of the one definition rule. While a diagnostic message is not required,[11] this typically results in a linker error.

In the case of the default constructor, the compiler will not generate a default constructor if a class is defined with any constructors. This is useful in many cases, but it is also useful to be able to have both specialized constructors and the compiler-generated default.

C++11 allows the explicit defaulting and deleting of these special member functions. For example, the following type explicitly declares that it is using the default constructor:

```
struct SomeType
{
   // ### The default constructor is explicitly stated.
   SomeType() = default;
   SomeType(OtherType value);
};
```

Alternatively, certain features can be explicitly disabled. For example, the following type is non-copyable:

```
struct NonCopyable
{
   NonCopyable & operator=(const NonCopyable&) = delete;
   NonCopyable(const NonCopyable&) = delete;
   NonCopyable() = default;
};
```

The = delete specifier can be used to prohibit calling any function, which can be used to disallow calling a member function with particular parameters. For example:

```
struct NoInt
{
   void f(double i);
   void f(int) = delete;
};
```

An attempt to call f() with an int will be rejected by the compiler, instead of performing a silent conversion to double. This can be generalized to disallow calling the function with any type other than double as follows:

```
struct OnlyDouble
{
   void f(double d);
   template<class T> void f(T) = delete;
};
```

8.7 Type long long int

In C++03, the largest integer type is long int. It is guaranteed to have at least as many usable bits as int. This resulted in long int having size of 64 bits on some popular implementations and 32 bits on others. C++11 adds a new integer type long long int to address this issue. It is guaranteed to be at least as large as a long int, and have no fewer than 64 bits. The type was originally introduced by C99 to the standard C, and most C++ compilers support it as an extension already.[12][13]

8.8 Static assertions

C++03 provides two methods to test assertions: the macro assert and the preprocessor directive #error. However, neither is appropriate for use in templates: the macro tests the assertion at execution-time, while the preprocessor directive tests the assertion during preprocessing, which happens before instantiation of templates. Neither is appropriate for testing properties that are dependent on template parameters.

The new utility introduces a new way to test assertions at compile-time, using the new keyword static_assert. The declaration assumes the following form:

```
static_assert (constant-expression, error-message);
```

Here are some examples of how static_assert can be used:

```
static_assert((GREEKPI > 3.14) && (GREEKPI < 3.15), "GREEKPI is
inaccurate!");

template<class T>
struct Check
{
   static_assert(sizeof(int) <= sizeof(T), "T is not big enough!");
};

template<class Integral>
Integral foo(Integral x, Integral y)
{
   static_assert(std::is_integral<Integral>::value,
      "foo() parameter must be an integral type.");
}
```

When the constant expression is false the compiler produces an error message. The first example represents an alternative to the preprocessor directive #error, in contrast in the second example the assertion is checked at every instantiation of the template class Check.

Static assertions are useful outside of templates as well. For instance, a particular implementation of an algorithm might depend on the size of a long long being larger than an int, something the standard does not guarantee. Such an assumption is valid on most systems and compilers, but not all.

8.9 Allow sizeof to work on members of classes without an explicit object

In C++03, the sizeof operator can be used on types and objects. But it cannot be used to do the following:

```
struct SomeType { OtherType member; };

// ### Does not work with C++03. Okay with C++11.
sizeof(SomeType::member);
```

This should return the size of OtherType. C++03 does not allow this, so it is a compile error. C++11 does allow it.

8.10 Control and query object alignment

C++11 allows variable alignment to be queried and controlled with alignof and alignas.

The alignof operator takes a type and returns the power of 2 byte boundary on which the type instances must be allocated (as a std::size_t). When given a reference type alignof returns the referenced type's alignment; for arrays it returns the element type's alignment.

The alignas specifier controls the memory alignment for a variable. The specifier takes a constant or a type; when supplied a type alignas(T) is short hand for alignas(alignof(T)). For example, to specify that a char array should be properly aligned to hold a float:

```
alignas(float) unsigned char c[sizeof(float)]
```

8.11 Allow garbage collected implementations

It is implementation-defined whether unreachable dynamically allocated objects are automatically reclaimed. However C++11 adds a few restrictions to implementations so that some behavior that would prevent garbage

collection to work is now disallowed. This includes in particular common ways to "hide" pointers from a possible garbage collector, like applying xor to it.

9 C++ standard library changes

A number of new features were introduced in the C++11 standard library. Many of these could have been implemented under the old standard, but some rely (to a greater or lesser extent) on new C++11 core features.

A large part of the new libraries was defined in the document C++ Standards Committee's Library Technical Report (called TR1), which was published in 2005. Various full and partial implementations of TR1 are currently available using the namespace std::tr1. For C++11 they were moved to namespace std. However, as TR1 features were brought into the C++11 standard library, they were upgraded where appropriate with C++11 language features that were not available in the initial TR1 version. Also, they may have been enhanced with features that were possible under C++03, but were not part of the original TR1 specification.

The committee intends to create a second technical report (called TR2) now that standardization of C++11 is complete. Library proposals which were not ready in time for C++11 will be put into TR2 or further technical reports.

9.1 Upgrades to standard library components

C++11 offers a number of new language features that the currently existing standard library components can benefit from. For example, most standard library containers can benefit from Rvalue reference based move constructor support, both for quickly moving heavy containers around and for moving the contents of those containers to new memory locations. The standard library components were upgraded with new C++11 language features where appropriate. These include, but are not necessarily limited to:

- Rvalue references and the associated move support
- Support for the UTF-16 encoding unit, and UTF-32 encoding unit Unicode character types
- Variadic templates (coupled with Rvalue references to allow for perfect forwarding)
- Compile-time constant expressions
- Decltype
- Explicit conversion operators
- Default/Deleted functions

Additionally, much time has passed since the previous C++ standard. A great deal of code using the standard library has been written; this has revealed portions of the standard libraries that could use some improvement. Among the many areas of improvement considered were standard library allocators. A new scope-based model of allocators was included in C++11 to supplement the previous model.

9.2 Threading facilities

While the C++11 language provides a memory model that supports threading, the primary support for actually using threading comes with the C++11 standard library.

The thread class std::thread is provided which takes a function object — and an optional series of arguments to pass to it — to run in the new thread. It is possible to cause a thread to halt until another executing thread

completes, providing thread joining support through the std::thread::join() member function. Access is provided, where feasible, to the underlying native thread object(s) for platform specific operations by the std::thread::native_handle() member function.

```
#include <thread>
void someFunction();
std::thread workerThread(someFunction);
workerThread.join() // ### Waits for the thread to finish.
```

Callable entities can be also be passed to the thread:

```
class SomeType
{
public:
   void operator()();
};

SomeType type;
std::thread workerThread(type);
```

For synchronization between threads, appropriate mutexes (std::mutex, std::recursive_mutex, etc.) and condition variables (std::condition_variable and std::condition_variable_any) are added to the library. These are accessible through RAII locks (std::lock_guard and std::unique_lock) and locking algorithms for easy use.

```
void someFunction()
{
   std::lock_guard<std::mutex> locker(someMutex);
   doSomething(strMember);
} // ### End of scope means the mutex will be released here.
```

For high-performance, low-level work, it is sometimes necessary to communicate between threads without the overhead of mutexes. This is achieved using atomic operations on memory locations. These can optionally specify the minimum memory visibility constraints required for an operation. Explicit memory barriers may also be used for this purpose.

The C++11 thread library also includes futures and promises for passing asynchronous results between threads, and std::packaged_task for wrapping up a function call that can generate such an asynchronous result. The futures proposal was criticized because it lacks a way to combine futures and check for the completion of one promise inside a set of promises.[14]

Further high-level threading facilities such as thread pools have been remanded to a future C++ technical report. They are not part of C++11, but their eventual implementation is expected to be built entirely on top of the thread library features.

The new std::async facility provides a convenient method of running tasks and tying them to a std::future. The user can choose whether the task is to be run asynchronously on a separate thread or synchronously on a thread that waits for the value. By default, the implementation can choose, which provides an easy way to take advantage of hardware concurrency without oversubscription, and provides some of the advantages of a thread pool for simple usages.

9.3 Tuple types

Tuples are collections composed of heterogeneous objects of pre-arranged dimensions. A tuple can be considered a generalization of a struct's member variables.

The C++11 version of the TR1 tuple type benefited from C++11 features like variadic templates. The TR1 version required an implementation-defined maximum number of contained types, and required substantial macro trickery to implement reasonably. By contrast, the implementation of the C++11 version requires no explicit implementation-defined maximum number of types. Though compilers will almost certainly have an

internal maximum recursion depth for template instantiation (which is normal), the C++11 version of tuples will not expose this value to the user.

Using variadic templates, the declaration of the tuple class looks as follows:

```
template <class ...Types> class tuple;
```

An example of definition and use of the tuple type:

```
typedef std::tuple <int, double, long &, const char *> test_tuple;
long lengthy = 12;
test_tuple proof (18, 6.5, lengthy, "Ciao!");

// ### Assign to 'lengthy' the value 18.
lengthy = std::get<0>(proof);
// ### Modify the tuple's fourth element.
std::get<3>(proof) = " Beautiful!";
```

It's possible to create the tuple proof without defining its contents, but only if the tuple elements' types possess default constructors. Moreover, it's possible to assign a tuple to another tuple: if the two tuples' types are the same, it is necessary that each element type possesses a copy constructor; otherwise, it is necessary that each element type of the right-side tuple is convertible to that of the corresponding element type of the left-side tuple or that the corresponding element type of the left-side tuple has a suitable constructor.

```
typedef std::tuple <int , double, string>
   tuple_1 t1;
typedef std::tuple <char, short , const char * >
   tuple_2 t2 ('X', 2, "Hola!");

// ### Ok, first two elements can be converted,
// ### the third one can be constructed from a 'const char *'.
t1 = t2 ;
```

Relational operators are available (among tuples with the same number of elements), and two expressions are available to check a tuple's characteristics (only during compilation):

- `std::tuple_size<T>::value` returns the number of elements in the tuple T,
- `std::tuple_element<I, T>::type` returns the type of the object number I of the tuple T.

9.4 Hash tables

Including hash tables (unordered associative containers) in the C++ standard library is one of the most recurring requests. It was not adopted in C++03 due to time constraints only. Although hash tables are less efficient than a balanced tree in the worst case (in the presence of many collisions), they perform better in many real applications.

Collisions are managed only through linear chaining because the committee didn't consider opportune to standardize solutions of open addressing that introduce quite a lot of intrinsic problems (above all when erasure of elements is admitted). To avoid name clashes with non-standard libraries that developed their own hash table implementations, the prefix "unordered" was used instead of "hash".

The new library has four types of hash tables, differentiated by whether or not they accept elements with the same key (unique keys or equivalent keys), and whether they map each key to an associated value. They correspond to the four existing binary-search-tree-based associative containers, with an unordered_ prefix.

Type of hash table	Associated values	Equivalent keys
`std::unordered_set`	No	No
`std::unordered_multiset`	No	Yes

std::unordered_map	Yes	No
std::unordered_multimap	Yes	Yes

New classes fulfill all the requirements of a container class, and have all the methods necessary to access elements: insert, erase, begin, end.

This new feature didn't need any C++ language core extensions (though implementations will take advantage of various C++11 language features), only a small extension of the header <functional> and the introduction of headers <unordered_set> and <unordered_map>. No other changes to any existing standard classes were needed, and it doesn't depend on any other extensions of the standard library.

9.5 Regular expressions

The new library, defined in the new header <regex>, is made of a couple of new classes:

- regular expressions are represented by instance of the template class std::regex;
- occurrences are represented by instance of the template class std::match_results.

The function std::regex_search is used for searching, while for 'search and replace' the function std::regex_replace is used which returns a new string. The algorithms std::regex_search and std::regex_replace take a regular expression and a string and write the occurrences found in the struct std::match_results.

Here is an example of the use of std::match_results:

```
// ### List of separator characters.
const char *reg_esp = "[ ,.\\t\\n;:]";

// ### this can be done using raw string literals:
// ### const char *reg_esp = R"([ ,.\t\n;:])";

// ### 'regex' is an instance of the template class
// ### 'basic_regex' with argument of type 'char'.
std::regex rgx(reg_esp);

// ### 'cmatch' is an instance of the template class
// ### 'match_results' with argument of type 'const char *'.
std::cmatch match;

const char *target = "Unseen University - Ankh-Morpork";

// ### Identifies all words of 'target' separated by
// ### characters of 'reg_esp'.
if( std::regex_search( target, match, rgx ) )
{
   // ### If words separated by specified characters are present.

   const size_t n = match.size();
   for( size_t a = 0; a < n; a++ )
   {
      std::string str( match[a].first, match[a].second );
      std::cout << str << "\n";
   }
}
```

Note the use of double backslashes, because C++ uses backslash as an escape character. The C++11 raw string feature could be used to avoid the problem.

The library <regex> requires neither alteration of any existing header (though it will use them where appropriate) nor an extension of the core language.

9.6 General-purpose smart pointers

In computer science, a smart pointer is an abstract data type that simulates a pointer while providing additional features, such as automatic garbage collection or bounds checking. These additional features are intended to reduce bugs caused by the misuse of pointers while retaining efficiency. Smart pointers typically keep track of the objects they point to for the purpose of memory management. They may also be used to manage other resources, such as network connections and file handles.

The misuse of pointers is a major source of bugs: the constant allocation, deallocation and referencing that must be performed by a program written using pointers introduces the risk that memory leaks will occur. Smart pointers try to prevent memory leaks by making the resource deallocation automatic: when the pointer (or the last in a series of pointers) to an object is destroyed, for example because it goes out of scope, the referenced object is destroyed too.

Several types of smart pointers exist. Some work with reference counting, others by assigning ownership of the object to a single pointer. If the language supports automatic garbage collection (for instance, Java or C#), then smart pointers are unnecessary for memory management, but may still be useful in managing other resources.

In C++, smart pointers may be implemented as a template class that mimics, by means of operator overloading, the behavior of traditional (raw) pointers, (e.g. dereferencing, assignment) while providing additional memory management algorithms.

Smart pointers can facilitate intentional programming by expressing the use of a pointer in the type itself. For example, if a C++ function returns a pointer, there is no way to know whether the caller should delete the memory pointed to when the caller is finished with the information.

```
// ### What should be done with the result?
some_type* ambiguous_function();
```

Traditionally, this has been solved with comments, but this can be error-prone. By returning an auto_ptr,

```
auto_ptr<some_type> obvious_function1();
```

the function makes explicit that the caller will take ownership of the result, and furthermore, that if the caller does nothing, no memory will be leaked. The auto_ptr was introduced with C++03.

1.1.1 unique_ptr

C++11 provides std::unique_ptr, defined in the header <memory>.

The copy constructor and assignment operators of std::auto_ptr do not actually copy the stored pointer. Instead, they transfer it, leaving the previous std::auto_ptr object empty. This was one way to implement strict ownership, so that only one auto_ptr object could own the pointer at any given time. This means that auto_ptr should not be used where copy semantics are needed.

C++11 provides support for move semantics; it allows for the explicit support of transferring values as a different operation from copying them. C++11 also provided support for explicitly preventing an object from being copied. Since std::auto_ptr already existed with its copy semantics, it could not be upgraded to be a move-only pointer without breaking backwards compatibility with existing code. Therefore, C++11 introduced a new pointer type: std::unique_ptr.

This pointer type has its copy constructor and assignment operator explicitly deleted; it cannot be copied. It can be moved using std::move, which allows one unique_ptr object to transfer ownership to another.

```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; // ### Compile error.
// ### Transfers ownership.
// ### Now p3 owns the memory and p1 is rendered invalid.
std::unique_ptr<int> p3 = std::move(p1);

p3.reset(); // ### Deletes the memory.
p1.reset(); // ### Does nothing.
```

std::auto_ptr is still available, but it is deprecated under C++11.

1.1.2 shared_ptr and weak_ptr

C++11 incorporates shared_ptr and weak_ptr, based on versions used by the Boost libraries. TR1 first introduced them to the standard, but C++11 gives them additional functionality in line with the Boost version.

std::shared_ptr represents reference counted ownership of a pointer. Each copy of the same shared_ptr owns the same pointer. That pointer will only be freed if all instances of the shared_ptr in the program are destroyed.

```
std::shared_ptr<int> p1(new int(5));
std::shared_ptr<int> p2 = p1; // ### Both now own the memory.

p1.reset(); // ### Memory still exists, due to p2.
p2.reset(); // ### Deletes the memory, since no one else owns the memory.
```

A std::shared_ptr uses reference counting, so circular references are potentially a problem. To break up cycles, std::weak_ptr can be used to access the stored object. The stored object will be deleted if the only references to the object are weak_ptr references. weak_ptr therefore does not ensure that the object will continue to exist, but it can ask for the resource.

```
std::shared_ptr<int> p1(new int(5));
std::weak_ptr<int> wp1 = p1; // ### p1 owns the memory.

{
   // ### Now p1 and p2 own the memory.
   std::shared_ptr<int> p2 = wp1.lock();
   if( p2 ) // ### Always check to see if the memory still exists
   {
      // ### Do something with p2
   }
} // ### p2 is destroyed. Memory is owned by p1.

p1.reset(); // ### Memory is deleted.

// ### Memory is gone, so we get an empty shared_ptr.
std::shared_ptr<int> p3 = wp1.lock();
if( p3 )
{
   // ### Will not execute this.
}
```

Operations that change the reference count, due to copying or destroying shared_ptr or weak_ptr objects, do not provoke data race conditions. This means that multiple threads can safely store shared_ptr or weak_ptr objects that reference the same object. This only protects the reference count itself; it does not protect the object being stored by the smart pointer.

The above only applies when multiple threads have their own shared_ptr instances that are referring to the same object. In cases where multiple threads are accessing the same shared_ptr instance, C++11 provides a number of atomic functions for accessing and manipulating the shared_ptr.

9.7 Extensible random number facility

The C standard library provides the ability to generate pseudorandom numbers through the function rand. However, the algorithm is delegated entirely to the library vendor. C++ inherited this functionality with no changes, but C++11 will provide a new method for generating pseudorandom numbers.

C++11's random number functionality is split into two parts: a generator engine that contains the random number generator's state and produces the pseudorandom numbers; and a distribution, which determines the range and mathematical distribution of the outcome. These two are combined to form a random number generator object.

Unlike the C standard rand, the C++11 mechanism will come with three base generator engine algorithms, `linear_congruential_engine`, `subtract_with_carry_engine` and `mersenne_twister_engine`.

C++11 will also provide a number of standard distributions: `uniform_int_distribution`, `uniform_real_distribution`, `bernoulli_distribution`, `binomial_distribution`, `geometric_distribution`, `negative_binomial_distribution`, `poisson_distribution`, `exponential_distribution`, `gamma_distribution`, `weibull_distribution`, `extreme_value_distribution`, `normal_distribution`, `lognormal_distribution`, `chi_squared_distribution`, `cauchy_distribution`, `fisher_f_distribution`, `student_t_distribution`, `discrete_distribution`, `piecewise_constant_distribution` and `piecewise_linear_distribution`.

The generator and distributions are combined as in the following example:

```
#include <random>
#include <functional>

std::uniform_int_distribution<int> distribution(0, 99);
std::mt19937 engine; // ### Mersenne twister MT19937
auto generator = std::bind(distribution, engine);
// ### Generate a uniform integral variate between 0 and 99.
int random = generator();
// ### Generate another sample directly
// ### using the distribution and the engine objects.
int random2 = distribution(engine);
```

9.8 Wrapper reference

A wrapper reference is obtained from an instance of the template class reference_wrapper. Wrapper references are similar to normal references (‘&’) of the C++ language. To obtain a wrapper reference from any object the function template ref is used (for a constant reference cref is used).

Wrapper references are useful above all for function templates, where references to parameters rather than copies are needed:

```
// ### This function will obtain a reference
// ### to the parameter 'r' and increment it.
void func (int &r)  { r++; }

// ### Template function.
template<class F, class P> void g (F f, P t)  { f(t); }
```

```
int main()
{
   int i = 0;
   // ### 'g<void (int &r), int>' is instantiated
   // ### then 'i' will not be modified.
   g (func, i);
   // ### Output -> 0
   std::cout << i << std::endl;

   // ### 'g<void(int &r),reference_wrapper<int>>' is instantiated
   // ### then 'i' will be modified.
   g (func, std::ref(i));
   std::cout << i << std::endl;  // ### Output -> 1
}
```

This new utility was added to the existing <utility> header and didn't need further extensions of the C++ language.

9.9 Polymorphic wrappers for function objects

Polymorphic wrappers for function objects are similar to function pointers in semantics and syntax, but are less tightly bound and can indiscriminately refer to anything which can be called (function pointers, member function pointers, or functors) whose arguments are compatible with those of the wrapper.

Through the example it is possible to understand its characteristics:

```
// ### Wrapper creation using template class 'function'.
std::function<int (int, int)> func;
// ### 'plus' is declared as 'template<class T> T plus( T, T ) ;'
// ### then 'add' is type 'int add( int x, int y )'.
std::plus<int> add;
// ### OK - Parameters and return types are the same.
func = add;

// ### NOTE: if the wrapper 'func' does not refer to any function,
// ### the exception 'std::bad_function_call' is thrown.
int a = func (1, 2);

std::function<bool (short, short)> func2 ;

// ### True because 'func2' has not yet been assigned a function.
if(!func2)
{

   bool adjacent(long x, long y);
   // ### OK - Parameters and return types are convertible.
   func2 = &adjacent ;

   struct Test
   {
      bool operator()(short x, short y);
   };
    Test car;
   // ### 'std::ref' is a template function that returns the wrapper
   // ### of member function 'operator()' of struct 'car'.
   func = std::ref(car);
}
```

```
// ### OK - Parameters and return types are convertible.
func = func2;
```

The template class function was defined inside the header <functional>, and didn't require any changes to the C++ language.

9.10 Type traits for metaprogramming

Metaprogramming consists of creating a program that creates or modifies another program (or itself). This can happen during compilation or during execution. The C++ Standards Committee has decided to introduce a library that allows metaprogramming during compilation through templates.

Here is an example of a meta-program, using the current C++03 standard: a recursion of template instances for calculating integer exponents:

```
template<int B, int N>
struct Pow
{
   // ### recursive call and recombination.
   enum{ value = B*Pow<B, N-1>::value };
};

template< int B >
struct Pow<B, 0>
{
   // ### ''N == 0'' condition of termination.
   enum{ value = 1 };
};
int quartic_of_three = Pow<3, 4>::value;
```

Many algorithms can operate on different types of data; C++'s templates support generic programming and make code more compact and useful. Nevertheless it is common for algorithms to need information on the data types being used. This information can be extracted during instantiation of a template class using type traits.

Type traits can identify the category of an object and all the characteristics of a class (or of a struct). They are defined in the new header <type_traits>.

In the next example there is the template function ‘elaborate’ that, depending on the given data types, will instantiate one of the two proposed algorithms (algorithm.do_it).

```
// ### First way of operating.
template< bool B > struct Algorithm
{
   template<class T1, class T2> static int do_it (T1 &, T2 &)  { /*...*/ }
};

// ### Second way of operating.
template<> struct Algorithm<true>
{
   template<class T1, class T2> static int do_it (T1, T2)  { /*...*/ }
};


// ### Instantiating 'elaborate' will automatically
// ### instantiate the correct way to operate.
template<class T1, class T2>
int elaborate (T1 A, T2 B)
```

```
{
   // ### Use the second way only if 'T1' is an integer and if 'T2' is
   // ### in floating point, otherwise use the first way.
   return (Algorithm<std::is_integral<T1>::value &&
      std::is_floating_point<T2>::value>::do_it( A, B ));
}
```

Through type traits, defined in header <type_transform>, it's also possible to create type transformation operations (static_cast and const_cast are insufficient inside a template).

This type of programming produces elegant and concise code; however the weak point of these techniques is the debugging: uncomfortable during compilation and very difficult during program execution.

9.11 Uniform method for computing the return type of function objects

Determining the return type of a template function object at compile-time is not intuitive, particularly if the return value depends on the parameters of the function. As an example:

```
struct Clear
{
   int    operator()(int) const;    // ### The parameter type is
   double operator()(double) const; // ### equal to the return type.
};

template <class Obj>
class Calculus
{
public:
   template<class Arg> Arg operator()(Arg& a) const
   {
      return member(a);
   }
private:
   Obj member;
};
```

Instantiating the class template Calculus<Clear>, the function object of calculus will have always the same return type as the function object of Clear. However, given class Confused below:

```
struct Confused
{
   double operator()(int) const;    // ### The parameter type is not
   int    operator()(double) const; // ### equal to the return type.
};
```

Attempting to instantiate Calculus<Confused> will cause the return type of Calculus to not be the same as that of class Confused. The compiler may generate warnings about the conversion from int to double and vice-versa.

TR1 introduces, and C++11 adopts, the template class std::result_of that allows one to determine and use the return type of a function object for every declaration. The object CalculusVer2 uses the std::result_of object to derive the return type of the function object:

```
template< class Obj >
class CalculusVer2
{
public:
   template<class Arg>
```

```
    typename std::result_of<Obj(Arg)>::type operator()(Arg& a) const
    {
       return member(a);
    }
private:
    Obj member;
};
```

In this way in instances of function object of CalculusVer2<Confused> there are no conversions, warnings, or errors.

The only change from the TR1 version of std::result_of is that the TR1 version allowed an implementation to fail to be able to determine the result type of a function call. Due to changes to C++ for supporting decltype, the C++11 version of std::result_of no longer needs these special cases; implementations are required to compute a type in all cases.

10 Features removed or deprecated

- The term sequence point, which is being replaced by specifying that either one operation is sequenced before another, or that two operations are unsequenced.[15]
- export[16]: Its current use is removed, but the keyword itself is still reserved, for potential future use.
- dynamic exception specifications[16] are deprecated. Compile time specification of non-exception throwing functions is available with the noexcept keyword (useful for optimization)
- std::auto_ptr is deprecated. Superseded by std::unique_ptr
- Function object base classes (std::unary_function, std::binary_function), adapters to pointers to functions and adapters to pointers to members, binder classes; these are all deprecated.

11 References

[1] http://video.google.com/videoplay?docid=5262479012306588324#

[2] "We have an international standard: C++0x is unanimously approved". Retrieved 12 August 2011.

[3] "Bjarne Stroustrup: A C++0x overview". Retrieved 30 June 2011.

[4] "ISO/IEC 14882:2011". ISO. 2 September 2011. Retrieved 3 September 2011.

[5] [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

[6] http://stackoverflow.com/a/4653479/734069

[7] Sutter, Alexandrescu "C++ coding standards" #15

[8] Gabriel Dos Reis and Bjarne Stroustrup (22 March 2010). "General Constant Expressions for System Programming Languages, Proceedings SAC '10".

[9] "Java Programming Language: Enhancements in JDK 5". Oracle.com.

[10] "Document no: N1968=06-0038- Lambda expressions and closures for C++". Open Standards.

[11] ISO/IEC (2003). ISO/IEC 14882:2003(E): Programming Languages - C++ §3.2 One definition rule [basic.def.odr] para. 3

[12] http://gcc.gnu.org/onlinedocs/gcc/Long-Long.html

[13] http://msdn.microsoft.com/en-us/library/s3f49ktz(VS.80).aspx

[14] Milewski, Bartosz (3 March 2009). "Broken promises–C++0x futures". Retrieved 24 January 2010.

[15] Caves, Jonathan (4 June 2007). "Update on the C++-0x Language Standard". Retrieved 25 May 2010.

[16] a b Sutter, Herb (3 March 2010). "Trip Report: March 2010 ISO C++ Standards Meeting". Retrieved 24 March 2010.

[17] Jeremy Manson and Brian Goetz (February 2004). "JSR 133 (Java Memory Model) FAQ". Retrieved 2010-10-18. "The Java Memory Model describes what behaviors are legal in multithreaded code, and how threads may interact through memory. It describes the relationship between variables in a program and the low-level details of storing and retrieving them to and from memory or registers in a real computer system. It does this in a way that can be implemented correctly using a wide variety of hardware and a wide variety of compiler optimizations."

[18] Goetz, Brian (2004-02-24). "Fixing the Java Memory Model, Part 1". Retrieved 2008-02-17.

[19] Boehm, Hans (2005-03-04). "Memory Model for Multthreaded C++". Retrieved 2008-02-17.

[20] Boehm, Hans. "Threads and memory model for C++". Retrieved 2008-02-17.