WIKIPEDIA

C++11

C++11 is a version of the standard for the programming language C++. It was approved by International Organization for Standardization (ISO) on 12 August 2011, replacing C++03,[1] superseded by C++14 on 18 August 2014[2] and later, by C++17. The name follows the tradition of naming language versions by the publication year of the specification, though it was formerly named *C++0x* because it was expected to be published before 2010.[3]

Although one of the design goals was to prefer changes to the libraries over changes to the core language,[4] C++11 does make several additions to the core language. Areas of the core language that were significantly improved include multithreading support, generic programming support, uniform initialization, and performance. Significant changes were also made to the C++ Standard Library, incorporating most of the C++ Technical Report 1 (TR1) libraries, except the library of mathematical special functions.[5]

C++11 was published as *ISO/IEC 14882:2011*[6] in September 2011 and is available for a fee. The working draft most similar to the published C++11 standard is N3337, dated 16 January 2012;[7] it has only editorial corrections from the C++11 standard.[8]

Contents

Design goals
Extensions to the C++ core language
- Core language runtime performance enhancements
  - Rvalue references and move constructors
  - constexpr – Generalized constant expressions
  - Modification to the definition of plain old data
- Core language build-time performance enhancements
  - Extern template
- Core language usability enhancements
  - Initializer lists
  - Uniform initialization
  - Type inference
  - Range-based for loop
  - Lambda functions and expressions
  - Alternative function syntax
  - Object construction improvement
  - Explicit overrides and final

- Null pointer constant
- Strongly typed enumerations
- Right angle bracket
- Explicit conversion operators
- Template aliases
- Unrestricted unions

Core language functionality improvements
- Variadic templates
- New string literals
- User-defined literals
- Multithreading memory model
- Thread-local storage
- Explicitly defaulted and deleted special member functions
- Type long long int
- Static assertions
- Allow sizeof to work on members of classes without an explicit object
- Control and query object alignment
- Allow garbage collected implementations
- Attributes

C++ standard library changes
- Upgrades to standard library components
- Threading facilities
- Tuple types
- Hash tables
- Regular expressions
- General-purpose smart pointers
- Extensible random number facility
- Wrapper reference
- Polymorphic wrappers for function objects
- Type traits for metaprogramming
- Uniform method for computing the return type of function objects

Improved C compatibility

Features originally planned but removed or not included

Features removed or deprecated

See also

References

External links

Design goals

The design committee attempted to stick to a number of goals in designing C++11:

- Maintain stability and compatibility with C++98 and possibly with C
- Prefer introducing new features via 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 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 (further support needed 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 most computer programmers will always be such, and because many beginners never widen their knowledge, limiting themselves to work in aspects of the language in which they specialize.[1]

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.

Core language runtime performance enhancements

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

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.[9] 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 unneeded deep copies that can happen implicitly when objects are passed by value. To illustrate the issue, consider that an std::vector<T> is, internally, a wrapper around a C-style array with a size. If an 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 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 an 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 an 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 an const std::vector<T>&, 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. To get an rvalue, the function template std::move() 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 (http://en.cppreference.com/w/cpp/container/vector/emplace_back) set of the C++ standard library methods.

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, in several places, the C++ specification requires using 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 invalid:

```
int get_five() {return 5;}

int some_value[get_five() + 7]; // Create an array of 12 integers. Ill-formed C++
```

This was not valid 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.[10] The above example can be rewritten as follows:

```
constexpr int get_five() {return 5;}

int some_value[get_five() + 7]; // Create an array of 12 integers. Valid C++11
```

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

Using constexpr on a function imposes some limits 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.

Before 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.

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, to allow objects of the type 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 objects 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 given invocation, the result is not a constant expression.

Modification to the definition of plain old data

In C++03, a class or struct must follow a number of rules 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. The C++03 standard has restrictions on what types are compatible with C or can be statically initialized despite there being no technical reason a compiler couldn't accept the program; if someone were to create a C++03 POD type and add a non-virtual member function, this type would no longer be a POD type, could not be statically initialized, and would be incompatible with C despite no change to the memory layout.

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 valid 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 all non-static data members to be 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 interoperate with C. Similarly, a class with public and private non-static data members would not be standard-layout, but it could be trivial and thus `memcpy`-able.

Core language build-time performance enhancements

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.

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.

Initializer lists

C++03 inherited the initializer-list feature from C. A struct or array is given a list of arguments in braces, 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;
};

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

This is very useful for static lists, or initializing a struct to some 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, such as:

```
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 via 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 these 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
```

Uniform initialization

C++03 has a number of problems with initializing types. Several ways to do this exist, and some produce different 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 needed. If no conversion exists, or only a narrowing conversion exists, the program is ill-formed. The initialization of `var2` invokes the constructor.

One can also do this:

```
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, which is still needed at times. 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. Thus this 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.

Type inference

In C++03 (and C), to use a variable, its type must be specified explicitly. 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. Thus, storing intermediates in variables is difficult, possibly needing knowledge of the internals of a given 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.[11][12] This creates a variable of the specific type of the initializer:

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

The type of some_strange_callable_type is simply whatever the particular template function override of std::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.

Further, the keyword decltype can be used to determine the type of 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 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)
```

which can be further compacted since "myvec" implements begin/end iterators:

```
for (auto& x : myvec)
```

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);
    auto a = v[0];        // a has type int
    decltype(v[0]) b = 1; // b has type const int&, the return type of
                          //   std::vector<int>::operator[](size_type) const
    auto c = 0;           // c has type int
    auto d = c;           // d has type int
    decltype(c) e;        // e has type int, the type of the entity named by c
    decltype((c)) f = c;  // f has type int&, because (c) is an lvalue
    decltype(0) g;        // g has type int, because 0 is an rvalue
}
```

Range-based for loop

C++11 extends the syntax of the for statement to allow for easy iteration over a range of elements:

```
int my_array[5] = {1, 2, 3, 4, 5};
// double the value of each element in my_array:
for (int& x : my_array) {
    x *= 2;
}
// similar but also using type inference for array elements
for (auto& 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 the standard library containers that have begin/end pairs will work with the range-based for statement.

Lambda functions and expressions

C++11 provides the ability to create anonymous functions, called lambda functions.[13] These are defined as follows:

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

The return type (-> int in this example) can be omitted as long as all return expressions return the same type. A lambda can optionally be a closure.

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 needed. However, as C++ grew more complex, it exposed several limits, especially regarding template function declarations. For example, in C++03 this is disallowed:

```
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 valid C++11
```

This is not valid 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;
}
```

Use of the keyword “auto” in this case is only part of the syntax and does not perform automatic type deduction.[14]

Object construction improvement

In C++03, constructors of a class are not allowed to call other constructors in an initializer list of that class. Each constructor must construct all of its class members itself or call a common member function, as follows:

```
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 (termed delegation). This allows constructors to utilize another constructor's behavior with a minimum of added code. Delegation has been used in other languages e.g., Java, Objective-C.

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 delegating 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. Thus, the C++11 compiler will generate code to perform the inheritance and the forwarding of the derived class to the base class. This is an all-or-nothing feature: either all of that base class's constructors are forwarded or none of them are. Also, an inherited constructor will be shadowed if it matches the signature of a constructor of the derived class, and restrictions exist for multiple inheritance: class constructors cannot be inherited from two classes that use constructors with the same signature.

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 this 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 assignment initialization shown above.

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);
};
```

Suppose the `Derived::some_func` is intended to replace the base class version. But instead, because it has a different signature, 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 {
  virtual void some_func(int) override; // ill-formed - doesn't override a base class method
};
```

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 indicate an error.

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 { };

struct Derived1 : Base1 { }; // ill-formed because the class Base1 has been marked final
struct Base2 {
    virtual void f() final;
};

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

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 for declarator attributes:

- they gain special meaning as attributes only when used in those specific trailing contexts (after all type specifiers, access specifiers, member declarations (for struct, class and enum types) and declarator specifiers, but before initialization or code implementation of each declarator in a comma-separated list of declarators);
- they do not alter the declared type signature and do not declare or override any new identifier in any scope;
- the recognized and accepted declarator attributes may be extended in future versions of C++ (some compiler-specific extensions already recognize added declarator attributes, to provide code generation options or optimization hints to the compiler, or to generate added data into the compiled code, intended for debuggers, linkers, and deployment of the compiled code, or to provide added system-specific security attributes, or to enhance reflection abilities at runtime, or to provide added binding information for interoperability with other programming languages and runtime systems; these extensions may take parameters between parentheses after the declarator attribute identifier; for ANSI conformance, these compiler-specific extensions should use the double underscore prefix convention).
- In any other location, they can be valid identifiers for new declarations (and later use if they are accessible).

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 using the preprocessor macro `NULL`, which commonly expands to either `((void*)0)` or `0`. C++ forbids implicit conversion from `void *` to other pointer types, thus removing the benefit of casting `0` to `void *`. As a consequence, only `0` is allowed 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_t 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(nullptr_t), not foo(int);
/*
  Note that foo(nullptr_t) will actually call foo(char *) in the example above using an implicit conversion,
  only if no other functions are overloading with compatible pointer types in scope.
  If multiple overloadings exist, the resolution will fail as it is ambiguous,
  unless there is an explicit declaration of foo(nullptr_t).

  In standard types headers for C++11, the nullptr_t  type should be declared as:
      typedef decltype(nullptr) nullptr_t;
  but not as:
      typedef int nullptr_t; // prior versions of C++ which need NULL to be defined as 0
      typedef void *nullptr_t; // ANSI C which defines NULL as ((void*)0)
*/
```

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. Further, the underlying integral type is implementation-defined; code that depends on the size of the enumeration is thus non-portable. Lastly, enumeration values are scoped to the enclosing scope. Thus, it is not possible for two separate enumerations in the same scope 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. Thus, they cannot be compared to integers either (the expression `Enumeration::Val4 == 101` gives a compile 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 this example:

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

With old-style enumerations the values are placed in the outer scope. With new-style enumerations they are placed within the scope of the enum class name. So in the above example, `Val1` is undefined, but `Enum2::Val1` is defined.

There is also a transitional syntax to allow old-style enumerations to provide explicit scoping, and the definition of the underlying type:

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

In this case the enumerator names are defined in the enumeration's scope (`Enum3::Val1`), but for backwards compatibility they are also placed in the enclosing scope.

Forward-declaring enums are also possible in C++11. Formerly, 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:

```
enum Enum1;                      // Invalid in C++03 and C++11; the underlying type cannot be determined.
enum Enum2 : unsigned int;       // Valid in C++11, the underlying type is specified explicitly.
enum class Enum3;                // Valid in C++11, the underlying type is int.
enum class Enum4 : unsigned int; // Valid in C++11.
enum Enum2 : unsigned short;     // Invalid in C++11, because Enum2 was formerly declared with a different underlying type.
```

Right angle bracket

C++03's parser defines “>>” as the right shift operator or stream extraction 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 around parameter expressions using the “>”, “>=” or “>>” binary operators:

```
template<bool Test> class SomeType;
std::vector<SomeType<1>2>> x1;  // Interpreted as a std::vector of SomeType<true>,
  // followed by "2 >> x1", which is not valid syntax for a declarator. 1 is true.
std::vector<SomeType<(1>2)>> x1;  // Interpreted as std::vector of SomeType<false>,
  // followed by the declarator "x1", which is valid C++11 syntax. (1>2) is false.
```

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 using those conversion functions in implicit conversions. However, language contexts that specifically need a boolean value (the conditions of if-statements and loops, and operands to the logical operators) count as explicit conversions and can thus use a bool conversion operator.

For example, this feature solves cleanly the safe bool issue.

Template aliases

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>
typedef SomeType<OtherType, Second, 5> TypedefName; // Invalid in C++03
```

This will not compile.

C++11 adds this ability with this 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 (*FunctionType)(double);       // Old style
using FunctionType = void (*)(double); // New introduced syntax
```

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 or destructor. C++11 lifts some of these restrictions.[3]

If a `union` member has a non trivial special member function, the compiler will not generate the equivalent member function for the `union` and it must be manually defined.

This is a simple example of a union permitted in C++11:

```
#include <new> // Needed for placement 'new'.

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

union U {
    int z;
    double w;
    Point p; // Invalid in C++03; valid in C++11.
    U() {} // Due to the Point member, a constructor definition is now needed.
    U(const Point& pt) : p(pt) {} // Construct Point object using initializer list.
    U& operator=(const Point& pt) { new(&p) Point(pt); return *this; } // Assign Point object using placement 'new'.
};
```

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

Core language functionality improvements

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

Variadic templates

In C++11, templates can take variable numbers of template parameters. This also allows the definition of type-safe variadic functions.

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 of undefined size and semantics. Neither literal type offers support for string literals with UTF-8, UTF-16, or any other kind of Unicode encodings.

The definition of the type `char` has been modified to explicitly express that it's at least the size needed 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 formerly defined as only the latter in the C++ standard itself, then relying on the C standard to guarantee at least 8 bits.

C++11 supports three Unicode encodings: UTF-8, UTF-16, and UTF-32. Along with the formerly 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.

Creating string literals for each of these encodings can be done thusly:

```
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[]` (note lower case 'u' prefix). The type of the third string is `const char32_t[]` (upper case 'U' prefix).

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

```
u8"This is a Unicode Character: \u2018."
u"This is a bigger Unicode Character: \u2018."
U"This is a Unicode Character: \U00002018."
```

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. Using 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.)"
```

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++03 code cannot create new literal modifiers.

By contrast, C++11 enables the user to define new kinds of literal modifiers that will construct objects based on the string of characters that the literal modifies.

Transformation of literals 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. All suffixes starting with any character except underscore (_) are reserved by the standard. Thus, all user-defined literals must have suffixes starting with an underscore (_).[15]

User-defined literals processing the raw form of the literal are defined via a literal operator, which is written as operator "". An example follows:

```
OutputType operator "" _mysuffix(const char * literal_string)
{
    // assumes that OutputType has a constructor that takes a const char *
    OutputType ret(literal_string);
    return ret;
}

OutputType some_variable = 1234_mysuffix;
// assumes that OutputType has a get_value() method that returns a double
assert(some_variable.get_value() == 1234.0)
```

The assignment statement `OutputType some_variable = 1234_mysuffix;` 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 via a variadic template:

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

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

This instantiates the literal processing function as `operator "" _tuffix<'1', '2', '3', '4'>()`. In this form, there is no null character terminating the string. The main purpose for doing this is to use C++11's `constexpr` keyword to ensure that the compiler will transform the literal entirely at compile time, assuming `OutputType` is a constexpr-constructible 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 an expression containing the sign as a unary prefix operator and the unsigned number.) There is no alternative template form:

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

OutputType some_variable = 1234_suffix; // Uses the 'unsigned long long' overload.
OutputType another_variable = 3.1416_suffix; // Uses the 'long double' overload.
```

In accord with the formerly mentioned new string prefixes, for string literals, these are used:

```
OutputType operator "" _ssuffix(const char     * string_values, size_t num_chars);
OutputType operator "" _ssuffix(const wchar_t  * string_values, size_t num_chars);
OutputType operator "" _ssuffix(const char16_t * string_values, size_t num_chars);
OutputType operator "" _ssuffix(const char32_t * string_values, size_t num_chars);
OutputType some_variable =   "1234"_ssuffix; // Uses the 'const char *' overload.
OutputType some_variable = u8"1234"_ssuffix; // Uses the 'const char *' overload.
OutputType some_variable =  L"1234"_ssuffix; // Uses the 'const wchar_t *'  overload.
OutputType some_variable =  u"1234"_ssuffix; // Uses the 'const char16_t *' overload.
OutputType some_variable =  U"1234"_ssuffix; // Uses the 'const char32_t *' overload.
```

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

Multithreading memory model

C++11 standardizes support for multithreaded programming.

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.)

The memory model defines when multiple threads may access the same memory location, and specifies when updates by one thread become visible to other threads.

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.

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 new`) that work on all classes, which the programmer can override.

However, there is very little control over creating 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 (ODR). While a diagnostic message is not required,[16] violations may result 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.[17] For example, this type explicitly declares that it is using the default constructor:

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

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

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

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;
};
```

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 supported it as an extension already.[18][19]

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 this 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 is similar to the preprocessor directive `#error`, although the preprocessor does only support integral types.[20] 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 also. For instance, a given 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.

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 this:

```
struct SomeType { OtherType member; };

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

This should return the size of OtherType. C++03 disallows this, so it is a compile error. C++11 allows it. It is also allowed for the alignof operator introduced in C++11.

Control and query object alignment

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

The alignof operator takes the 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 shorthand 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)]
```

Allow garbage collected implementations

Prior C++ standards provided for programmer-driven garbage collection via set_new_handler, but gave no definition of object reachability for the purpose of automatic garbage collection. C++11 defines conditions under which pointer values are "safely derived" from other values. An implementation may specify that it operates under *strict pointer safety*, in which case pointers that are not derived according to these rules can become invalid.

Attributes

C++11 provides a standardized syntax for compiler/tool extensions to the language. Such extensions were traditionally specified using #pragma directive or vendor-specific keywords (like __attribute__ for GNU and __declspec for Microsoft). With the new syntax, added information can be specified in a form of an attribute enclosed in double square brackets. An attribute can be applied to various elements of source code:

```
int [[attr1]] i [[attr2, attr3]];
```

```
[[attr4(arg1, arg2)]] if (cond)
{
    [[vendor::attr5]] return i;
}
```

In the example above, attribute attr1 applies to the type of variable i, attr2 and attr3 apply to the variable itself, attr4 applies to the if statement and vendor::attr5 applies to the return statement. In general (but with some exceptions), an attribute specified for a named entity is placed after the name, and before the entity otherwise, as shown above, several attributes may be listed inside one pair of double square brackets, added arguments may be provided for an attribute, and attributes may be scoped by vendor-specific attribute namespaces.

It is recommended that attributes have no language semantic meaning and do not change the sense of a program when ignored. Attributes can be useful for providing information that, for example, helps the compiler to issue better diagnostics or optimize the generated code.

C++11 provides two standard attributes itself: noreturn to specify that a function does not return, and carries_dependency to help optimizing multi-threaded code by indicating that function arguments or return value carry a dependency.

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.

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

Further, much time has passed since the prior C++ standard. Much code using the standard library has been written. This has revealed parts of the standard libraries that could use some improving. 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 prior model.

Threading facilities

While the C++03 language provides a memory model that supports threading, the primary support for actually using threading comes with the C++11 standard library.

A 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 via 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.

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 via Resource Acquisition Is Initialization (RAII) locks (std::lock_guard and std::unique_lock) and locking algorithms for easy use.

For high-performance, low-level work, communicating between threads is sometimes needed without the overhead of mutexes. This is done using atomic operations on memory locations. These can optionally specify the minimum memory visibility constraints needed 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.[21]

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.

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. To implement reasonably, the TR1 version required an implementation-defined maximum number of contained types, and substantial macro trickery. By contrast, the implementation of the C++11 version requires no explicit implementation-defined maximum number of types. Though compilers will 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!");

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

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, each element type must possess a copy constructor; otherwise, each element type of the right-side tuple must be 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!");
t1 = t2; // Ok, first two elements can be converted,
         // the third one can be constructed from a 'const char *'.
```

Just like std::make_pair for std::pair, there exists std::make_tuple to automatically create std::tuples using type deduction and auto helps to declare such a tuple. std::tie creates tuples of lvalue references to help unpack tuples. std::ignore also helps here. See the example:

```
auto record = std::make_tuple("Hari Ram", "New Delhi", 3.5, 'A');
std::string name ; float gpa ; char grade ;
std::tie(name, std::ignore, gpa, grade) = record ; // std::ignore helps drop the place name
std::cout << name << ' ' << gpa << ' ' << grade << std::endl ;
```

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.

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 via *linear chaining* because the committee didn't consider it to be 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

The new classes fulfill all the requirements of a container class, and have all the methods needed 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.

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`:

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

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

std::regex rgx(reg_esp); // 'regex' is an instance of the template class
                         // 'basic_regex' with argument of type 'char'.
std::cmatch match; // 'cmatch' is an instance of the template class
                   // 'match_results' with argument of type 'const char *'.
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. In POSIX C, regular expressions are also available the C POSIX library#regex.h.

General-purpose smart pointers

C++11 provides `std::unique_ptr`, and improvements to `std::shared_ptr` and `std::weak_ptr` from TR1. `std::auto_ptr` is deprecated.

Extensible random number facility

The C standard library provides the ability to generate pseudorandom numbers via the function `rand`. However, the algorithm is delegated entirely to the library vendor. C++ inherited this functionality with no changes, but C++11 provides 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 also provides 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 this 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);
int random = generator(); // Generate a uniform integral variate between 0 and 99.
int random2 = distribution(engine); // Generate another sample directly using the distribution and the engine objects.
```

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

    g (func, std::ref(i)); // 'g<void(int &r),reference_wrapper<int>>' is instantiated
                           // then 'i' will be modified.
    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.

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.

An example can clarify its characteristics:

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

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

std::function<bool (short, short)> func2 ;
if (!func2) { // True because 'func2' has not yet been assigned a function.

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

    struct Test {
        bool operator()(short x, short y);
    };
    Test car;
    func = std::ref(car); // 'std::ref' is a template function that returns the wrapper
                          // of member function 'operator()' of struct 'car'.
}
func = func2; // OK - Parameters and return types are convertible.
```

The template class function was defined inside the header <functional>, without needing any change to the C++ language.

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 compiling via templates.

Here is an example of a meta-program, using the 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 ) ;
}
```

Via *type traits*, defined in header `<type_traits>`, 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.

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.

Improved C compatibility

For compatibility with C, from C99, these were added:[22]

- Preprocessor:[23]
  - variadic macros,
  - concatenation of adjacent narrow/wide string literals,
  - _Pragma() – equivalent of #pragma.
- long long – integer type that is at least 64 bits long.
- __func__ – macro evaluating to the name of the function it is in.
- Headers:
  - cstdbool (stdbool.h),
  - cstdint (stdint.h),
  - cinttypes (inttypes.h).

Features originally planned but removed or not included

Heading for a separate TR:

- Modules (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3347.pdf)
- Decimal types
- Math special functions

Postponed:

- Concepts
- More complete or required garbage collection support
- Reflection
- Macro scopes

Features removed or deprecated

The term sequence point was removed, being replaced by specifying that either one operation is sequenced before another, or that two operations are unsequenced.[24]

The former use of the keyword `export` was removed.[25] The keyword itself remains, being reserved for potential future use.

Dynamic exception specifications are deprecated.[25] Compile-time specification of non-exception-throwing functions is available with the `noexcept` keyword, which is useful for optimization.

`std::auto_ptr` is deprecated, having been 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, and binder classes are all deprecated.

See also

- C11
- C++98
- C++03
- C++14
- C++17

References

1. "We have an international standard: C++0x is unanimously approved" (http://herbsutter.com/2011/08/12/we-have-an-international-standard-c0x-is-unanimously-approved/). Retrieved 12 August 2011.
2. Sutter, Herb (August 18, 2014), *We have C++14!* (https://isocpp.org/blog/2014/08/we-have-cpp14), retrieved 2014-08-18
3. Stroustrup, Bjarne. "C++11 FAQ" (http://www.stroustrup.com/C++11FAQ.html). *stroustrup.com*.
4. "C++11 Overview: What specific design goals guided the committee?" (https://isocpp.org/wiki/faq/cpp11#cpp11-specific-goals). *Standard C++*.
5. "Bjarne Stroustrup: A C++0x overview" (https://www.research.ibm.com/arl/seminar/media/stroustrup.pdf) (PDF). Retrieved 30 June 2011.
6. "ISO/IEC 14882:2011" (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372). ISO. 2 September 2011. Retrieved 3 September 2011.
7. "Working Draft, Standard for Programming Language C++" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf) (PDF).
8. "The Standard" (http://isocpp.org/std/the-standard). Retrieved 2012-11-02.
9. Sutter, Alexandrescu "C++ coding standards" #15
10. Gabriel Dos Reis; Bjarne Stroustrup (22 March 2010). "General Constant Expressions for System Programming Languages, Proceedings SAC '10" (http://www.stroustrup.com/sac10-constexpr.pdf) (PDF).
11. Jaakko Järvi; Bjarne Stroustrup; Douglas Gregor; Jeremy Siek (April 28, 2003). "Decltype and auto, Programming Language C++, Document no: N1478=03-0061" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1478.pdf) (PDF).
12. Roger Orr (June 2013). " "Auto – A Necessary Evil?" Overload Journal #115" (http://accu.org/index.php/journals/1859).

13. "Document no: N1968=06-0038- Lambda expressions and closures for C++" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf) (PDF). Open Standards.
14. "auto specifier (since C++11) - cppreference.com" (http://en.cppreference.com/w/cpp/language/auto). *en.cppreference.com*.
15. This caused a conflict with the proposed use (common in other languages) of the underscore for digit grouping in numeric literals such as integer literals, so C++14 instead uses the apostrophe (as an upper comma) for grouping.Daveed Vandevoorde (2012-09-21). "N3448: Painless Digit Separation" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3448.pdf) (PDF)., Lawrence Crowl (2012-12-19). "N3499: Digit Separators" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3499.html).
16. ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages – C++ §3.2 One definition rule [basic.def.odr]* para. 3
17. "Defaulted and Deleted Functions – ISO/IEC JTC1 SC22 WG21 N2210 = 07-0070 – 2007-03-11" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2210.html#%22default%22).
18. "Using the GNU Compiler Collection (GCC): Long Long" (https://gcc.gnu.org/onlinedocs/gcc/Long-Long.html). *gcc.gnu.org*.
19. Data Type Ranges (C++) (http://msdn.microsoft.com/en-us/library/s3f49ktz(VS.80).aspx)
20. Samuel P. Harbison III, Guy L. Steele Jr.: "C – A Reference Manual", 5th edition, p.251
21. Milewski, Bartosz (3 March 2009). "Broken promises–C++0x futures" (http://bartoszmilewski.wordpress.com/2009/03/03/broken-promises-c0x-futures/). Retrieved 24 January 2010.
22. "Clang - C++98, C++11, and C++14 Status" (http://clang.llvm.org/cxx_status.html). Clang.llvm.org. 2013-05-12. Retrieved 2013-06-10.
23. "Working draft changes for C99 preprocessor synchronization" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1653.htm). *www.open-std.org*.
24. Caves, Jonathan (4 June 2007). "Update on the C++-0x Language Standard" (http://blogs.msdn.com/b/vcblog/archive/2007/06/04/update-on-the-c-0x-language-standard.aspx). Retrieved 25 May 2010.
25. Sutter, Herb (3 March 2010). "Trip Report: March 2010 ISO C++ Standards Meeting" (http://herbsutter.com/2010/03/13/trip-report-march-2010-iso-c-standards-meeting/). Retrieved 24 March 2010.

External links

- The C++ Standards Committee (http://www.open-std.org/jtc1/sc22/wg21/)
- C++0X: The New Face of Standard C++ (http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=216)
- Herb Sutter's blog coverage of C++11 (http://herbsutter.wordpress.com/)
- Anthony Williams' blog coverage of C++11 (http://www.justsoftwaresolutions.co.uk/cplusplus/)
- A talk on C++0x given by Bjarne Stroustrup at the University of Waterloo (http://www.csclub.uwaterloo.ca/media/C++0x%20-%20An%20Overview.html)
- The State of the Language: An Interview with Bjarne Stroustrup (15 August 2008) (http://www.devx.com/SpecialReports/Article/38813/0/page/1)
- Wiki page to help keep track of C++ 0x core language features and their availability in compilers (http://wiki.apache.org/stdcxx/C++0xCompilerSupport)
- Online C++11 standard library reference (http://en.cppreference.com)
- Online C++11 compiler (http://stacked-crooked.com/)
- Bjarne Stroustrup's C++11 FAQ (http://www.stroustrup.com/C++11FAQ.html)
- More information on C++11 features:range-based for loop,why auto_ptr is deprecated,etc. (https://corecplusplustutorial.com)

Retrieved from "https://en.wikipedia.org/w/index.php?title=C%2B%2B11&oldid=827709009"