C++11

From Wikipedia, the free encyclopedia

C++11, also formerly known as **C++0x** (pronounced "see plus plus oh ex"),[1] is the name of the most recent iteration of the C++ programming language, replacing C++03, approved by the ISO as of 12 August 2011.[2] The name is derived from the tradition of naming language versions by the date 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 most recent working draft available is (N3242 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf)) dated 28 February 2011.

Contents

Candidate changes for the impending standard update

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

In the development of every utility of the new 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 only useful to specific applications;
- Increase type safety by providing safer alternatives to current, 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] Additionally, considering the vastness of C++ and its usage (including areas of application and programming styles), even the most experienced programmers can become beginners in a new programming paradigm.

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 4 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 only mentioned in the group which primarily represents that feature.

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) - though in some cases temporaries could have been modified, and it was even considered to be a useful loophole (for the former, see Sutter, Alexandrescu "C++ coding standards" #15). 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 only be stored 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 simply 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. Because the temporary's pointer is allowed to be cleared, its memory is not deleted when it goes out of scope, therefore a deep copy is not required. Moreover, this is a safe and invisible operation because the temporary will never again be used.

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's declared as such; in order to get an rvalue, the function template `std::move<T>()` should be used. Rvalue references can also only be modified 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.

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, constant expressions have always ended whenever a function call or object constructor was encountered. So a piece of code as simple as this is illegal:

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

int some_value[get_five() + 7]; //create an array of 12 integers. Illeg
```

This is not legal C++, because `get_five() + 7` is not a constant expression. The 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. 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. Legal
```

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

The use of `constexpr` on a function imposes very strict limitations on what that function can do. First, the function must have a non-void return type. Second, the function contents must be of the form: `return` *`expr`*. Third, `expr` must be a constant expression, after argument substitution. This constant expression may only call other functions defined as `constexpr`, or it may use other constant expression data variables. Lastly, a function with this label cannot be called until it is defined in this translation unit.

Variables can also be defined as constant expression values:

```
constexpr double acceleration_due_to_gravity = 9.8;
constexpr double moon_gravity = acceleration_due_to_gravity / 6.0;
```

Constant expression data variables are implicitly const. They can only store the results of constant expressions or constant expression constructors.

In order to construct constant expression data values from user-defined types, constructors can also be declared with `constexpr`. A constant expression constructor must be defined before its use in the translation unit, as with constant expression functions. It must have an empty function body. It must initialize its members with constant expressions. The destructors for such types should be trivial.

Copying constexpr constructed types should also be defined as a `constexpr`, 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 fit the definition for function constant expressions. This allows the compiler to copy classes at compile time, perform operations on them, etc.

A constant expression function or constructor can be called with non-constexpr parameters. Just as a constexpr integer literal can be assigned to a non-constexpr variable, so too can a constexpr function be called with non-constexpr parameters, and the results stored in non-constexpr variables. The keyword only allows for the possibility of compile-time constancy when all members of an expression are constexpr.

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. 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.

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

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.

A class or struct is standard-layout, by definition, provided:

1. It has no virtual functions
2. It has no virtual base classes
3. It has no base classes of the same type as the first defined non-static data member
4. All its non-static data members have the same access control (public, private, protected)
5. All its non-static data members, including any in its base classes, are in the same one class in the hierarchy
6. The above rules also apply to all the base classes and to all non-static data members in the hierarchy

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 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;
};

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

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 only allows initializer-lists 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 only be initially constructed statically by the C++11 compiler 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" };
```

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 {"SomeName", 4}; //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.

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(&some_function, _2, _1, s
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.begin(); itr != myvec
```

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 `typedef`s 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) cons
    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 nam
    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

In C++03, iterating over the elements of a list requires a lot of code. Other languages like C# and Java have shortcuts that allow one to write a simple “foreach” statement that automatically walks the list from start to finish.

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 a `begin()` and `end()` function defined for it that returns iterators. All of the standard library containers that have begin/end pairs will work with the range-based for statement.

Lambda functions and expressions

Main article: Anonymous function#C++

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

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

The return type is implicit; it returns the type of the return expression (`decltype(x+y)`). The return type of a lambda can be omitted so long as all `return` expressions return the same type.

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;} /
```

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
```

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)
```

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.

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 only be initialized 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.

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 {
    virtual void some_func(int) override; // ill-formed because it does
};
```

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 { };

struct Derived1 : Base1 { }; // ill-formed because the class Base1 has

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

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

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 only gain special meaning when used in those specific contexts. In any other location, they can be valid identifiers.

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, only allowing `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);
```

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 will allow 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 never implementation-defined. The default, as in the above case, is `int`, but a different type can be explicitly specified as follows:

```
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 standard 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), but for

backwards compatibility, enumerator names are also placed in the enclosing scope.

Declaring enums is also possible in C++11. Previously, enum types could not be 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 declared:

```
enum Enum1;                    //Illegal in C++ and C++11; the underlyin
enum Enum2 : unsigned int;     //Legal in C++11, the underlying type is
enum class Enum3;              //Legal in C++11, the underlying type is
enum class Enum4 : unsigned int; //Legal C++11.
enum Enum2 : unsigned short;   //Illegal in C++11, because Enum2 was pre
```

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 will improve 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;
std::vector<SomeType<1>2>> x1;  // Interpreted as a std::vector of Some
// which is not legal syntax. 1 is true.
std::vector<SomeType<(1>2)>> x1;  // Interpreted as std::vector of Some
// which is legal C++11 syntax. (1>2) is false.
```

Explicit conversion operators

C++03 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.

Template aliases

In C++03, it is only possible to define a typedef 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; //Illegal in C++
```

This will not compile.

C++11 will add 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
```

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 will lift 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;
    Point p;  // Illegal in C++; point has a non-trivial constructor.
    U() { new( &p ) Point(); } // No nontrivial member functions are im
                               // if required they are instead deleted
};
```

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

Identifiers with special meaning

The identifiers `override` and `final` have a special meaning when used in a certain context, but can otherwise be used as normal identifiers.

Core language functionality improvements

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

Variadic templates

Main article: 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. 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 will add 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 will allow 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 will provide 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 only ends when `)delimiter"` is reached. The string `delimiter` can be any string up to 16 characters in length. 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.)"
```

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 will also include 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 only be processed 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);

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

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_c
OutputType operator "" _suffix(const wchar_t * string_values, size_t nu
OutputType operator "" _suffix(const char16_t * string_values, size_t n
OutputType operator "" _suffix(const char32_t * string_values, size_t n

OutputType some_variable = "1234"_suffix;      //Calls the const char *
OutputType some_variable = u8"1234"_suffix;    //Calls the const char *
OutputType some_variable = L"1234"_suffix;     //Calls the const wchar_
OutputType some_variable = u"1234"_suffix;     //Calls the const char16_
OutputType some_variable = U"1234"_suffix;     //Calls the const char32_
```

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

Multitasking memory model

See also: Memory model (computing)

The C++ standard committee plans to standardize support for multithreaded programming.

There are two parts involved: defining a memory model which will allow multiple threads to co-exist in a program, and defining support for interaction between threads. The second part will be provided via library facilities. (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) has been proposed for the next standard. Thread local storage will be 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 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,[5] this typically results in a linker error.[*citation needed*]

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 will allow 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 {
    SomeType() = default; //The default constructor is explicitly state
    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;
};
```

Type `long long int`

In C++03, the largest integral 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 integral type `long long int` to address this issue. It is guaranteed to be at least as large as a `long int`, and have no less than 64 bits. The type was originally introduced by C99 to the standard C, and most C++ compilers support it as an extension already.[6][7]

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 inaccu
```

```
template<class T>
```

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

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.

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; };

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

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

Allow garbage collected implementations

It is implementation-defined whether unreachable dynamically allocated objects are automatically reclaimed.

C++ standard library changes

A number of new features will be introduced in the C++11 standard library. Many of these can be implemented under the current standard, but some rely (to a greater or lesser extent) on new C++11 core features.

A large part of the new libraries is 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 will be moved to namespace `std`. However, as TR1 features are brought into the C++11 standard library, they are upgraded where appropriate with C++11 language features that were not available in the initial TR1 version. Also, they may be 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) after the standardization of C++11 is complete. Library proposals which are not ready in time for C++11 will be put into TR2 or further technical reports.

The following proposals are under way for C++11.

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 will be 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 C++ was standardized. 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 being considered are standard library allocators. A new scope-based model of allocators will be included in the C++11 to supplement the current model.

Threading facilities

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

A thread class (`std::thread`) will be provided which will take a function object — and an optional series of arguments to pass to it — to run in the new thread. It will be possible to cause a thread to halt until another executing thread completes, providing thread joining support through the `std::thread::join()` member function. Access will also be 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`) will be added to the library. This will be accessible through RAII locks (`std::lock_guard` and `std::unique_lock`) and locking algorithms for easy use.

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 will also include 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.[8]

Further high-level threading facilities such as thread pools have been remanded to a future C++ technical report. They will not be a 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 will benefit 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!");

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

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

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 elements' number of 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 the current standard due to time constraints only. Although this solution is less efficient than a balanced tree in the worst case (in the presence of many collisions), it performs better in many real applications.

Collisions will be managed only through *linear chaining* because the committee doesn'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" will be used instead of "hash".

The new utility will have 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.

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 utility doesn't need any C++ language core extensions (though the implementation 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 are needed, and it doesn't depend on any other extensions of the standard library.

Regular expressions

Many more or less standardized libraries were created to manage regular expressions. Since the use of these algorithms is very common, the standard library will include them using all potentialities of an object oriented language.

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 on 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 cla
                          // '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.

General-purpose smart pointers

Main article: C++ Smart Pointers

C++11 provides `std::unique_ptr`, as well as 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 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 generator engine algorithms, each with its own strengths and weaknesses:

Class template	Integral/floating-point	Quality	Speed	Size of state
`linear_congruential`	Integral	Medium	Medium	1
`subtract_with_carry`	Both	Medium	Fast	25
`mersenne_twister`	Integral	Good	Fast	624

C++11 will also provide a number of standard distributions: `uniform_int_distribution`, `bernoulli_distribution`, `geometric_distribution`, `poisson_distribution`, `binomial_distribution`, `uniform_real_distribution`, `exponential_distribution`, `normal_distribution`, and `gamma_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);
int random = generator();  // Generate a uniform integral variate betwe
```

```
int random2 = distribution(engine); // Generate another sample directly
```

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 incre
void f (int &r)  { r++; }

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

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

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

This new utility will be added to the existing `<utility>` header and doesn'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.

Through the example it is possible to understand 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
                 // 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
                      // the exception 'std::bad_function_call' is thro

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

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

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

The template class `function` will be defined inside the header `<functional>`, and doesn't require any changes 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 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
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_po
}
```

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.

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);  // The parameter type is
    double operator()(double);  // 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);  // The parameter type is NOT
    int    operator()(double);  // 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.

Features planned but removed or not included

Heading for a separate TR:

- Modules (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2316.pdf)
- Decimal Types
- Math Special Functions

Postponed:

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

Features to be 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.[9]
- export[10]
- dynamic exception specifications[10]. Compile time specification of non-exception throwing functions is available with the noexcept keyword (useful for optimization)
- std::auto_ptr 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.

See also

- C++ Technical Report 1
- C99, latest standard for the C programming language
- C1X, the planned new C standard

References

1. ^ http://video.google.com/videoplay?docid=52624790123065883 24#
2. ^ "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/) . http://herbsutter.com/2011/08/12/we-have-an-international-standard-c0x-is-unanimously-approved/. Retrieved 12 August 2011.
3. ^ "Bjarne Stroustrup: A C++0x overview" (http://www.research.ibm.com/arl/seminar/media/stroustrup.pdf) . http://www.research.ibm.com/arl/seminar/media/stroustrup.pdf. Retrieved 30 June 2011.
4. ^ "ISO/IEC 14882:2011" (http://www.iso.org/iso/iso_catalogue/catalogue_tc

/catalogue_detail.htm?csnumber=50372) . ISO. 2 September 2011. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372. Retrieved 3 September 2011.
5. ^ ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §3.2 One definition rule [basic.def.odr]* para. 3
6. ^ http://gcc.gnu.org/onlinedocs/gcc/Long-Long.html
7. ^ http://msdn.microsoft.com/en-us/library/s3f49ktz(VS.80).aspx
8. ^ Milewski, Bartosz (3 March 2009). "Broken promises–C++0x futures" (http://bartoszmilewski.wordpress.com/2009/03/03/broken-promises-c0x-futures/) . http://bartoszmilewski.wordpress.com/2009/03/03/broken-promises-c0x-futures/. Retrieved 24 January 2010.
9. ^ 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) . http://blogs.msdn.com/b/vcblog/archive/2007/06/04/update-on-the-c-0x-language-standard.aspx. Retrieved 25 May 2010.
10. ^ ***a*** ***b*** 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/) . http://herbsutter.com/2010/03/13/trip-report-march-2010-iso-c-standards-meeting/. Retrieved 24 March 2010.

Further reading

C++ Standards Committee papers

- ^ Doc No. 1401 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1401.pdf) : Jan Kristoffersen (21 October 2002) *Atomic operations with multi-threaded environments*
- ^ Doc No. 1402 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1402.html) : Doug Gregor (22 October 2002) *A Proposal to add a Polymorphic Function Object Wrapper to the Standard Library*
- ^ Doc No. 1403 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1403.pdf) : Doug Gregor (8 November 2002) *Proposal for adding tuple types into the standard library*
- ^ Doc No. 1424 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1424.htm) : John Maddock (3 March 2003) *A Proposal to add Type Traits to the Standard Library*
- ^ Doc No. 1429 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1429.htm) : John Maddock (3 March 2003) *A Proposal to add Regular Expression to the Standard Library*
- ^ Doc No. 1449 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1449.pdf) : B. Stroustrup, G. Dos Reis, Mat Marcus, Walter E. Brown, Herb Sutter (7 April 2003) *Proposal to add template aliases to C++*
- ^ Doc No. 1450 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1450.html) : P. Dimov, B. Dawes, G. Colvin (27 March 2003) *A Proposal to Add General Purpose Smart Pointers to the Library Technical Report (Revision 1)*
- ^ Doc No. 1452 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1452.html) : Jens Maurer (10 April 2003) *A Proposal to Add an Extensible Random Number Facility to the Standard Library (Revision 2)*
- ^ Doc No. 1453 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1453.html) : D. Gregor, P. Dimov (9 April 2003) *A proposal to add a*

reference wrapper to the standard library (revision 1)
- ^ Doc No. 1454 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1454.html) : Douglas Gregor, P. Dimov (9 April 2003) *A uniform method for computing function object return types (revision 1)*
- ^ Doc No. 1456 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html) : Matthew Austern (9 April 2003) *A Proposal to Add Hash Tables to the Standard Library (revision 4)*
- ^ Doc No. 1471 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1471.pdf) : Daveed Vandevoorde (18 April 2003) *Reflective Metaprogramming in C++*
- ^ Doc No. 1676 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1676.htm) : Bronek Kozicki (9 September 2004) *Non-member overloaded copy assignment operator*
- ^ Doc No. 1704 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1704.pdf) : Douglas Gregor, Jaakko Järvi, Gary Powell (10 September 2004) *Variadic Templates: Exploring the Design Space*
- ^ Doc No. 1705 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1705.pdf) : J. Järvi, B. Stroustrup, D. Gregor, J. Siek, G. Dos Reis (12 September 2004) *Decltype (and auto)*
- ^ Doc No. 1717 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1717.pdf) : Francis Glassborow, Lois Goldthwaite (5 November 2004) *explicit class and default definitions*
- ^ Doc No. 1719 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1719.pdf) : Herb Sutter, David E. Miller (21 October 2004) *Strongly Typed Enums (revision 1)*
- ^ Doc No. 1720 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html) : R. Klarer, J. Maddock, B. Dawes, H. Hinnant (20 October 2004) *Proposal to Add Static Assertions to the Core Language (Revision 3)*
- ^ Doc No. 1757 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html) : Daveed Vandevoorde (14 January 2005) *Right Angle Brackets (Revision 2)*
- ^ Doc No. 1811 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1811.pdf) : J. Stephen Adamczyk (29 April 2005) *Adding the long long type to C++ (Revision 3)*
- ^ Doc No. 1815 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1815.html) : Lawrence Crowl (2 May 2005) *ISO C++ Strategic Plan for Multithreading*
- ^ Doc No. 1827 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1827.htm) : Chris Uzdavinis, Alisdair Meredith (29 August 2005) *An Explicit Override Syntax for C++*
- ^ Doc No. 1834 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1834.html) : Detlef Vollmann (24 June 2005) *A Pleading for Reasonable Parallel Processing Support in C++*
- ^ Doc No. 1836 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf) : ISO/IEC DTR 19768 (24 June 2005) *Draft Technical Report on C++ Library Extensions*
- ^ Doc No. 1886 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1886.pdf) : Gabriel Dos Reis, Bjarne Stroustrup (20 October 2005) *Specifying C++ concepts*
- ^ Doc No. 1891 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1891.pdf) : Walter E. Brown (18 October 2005) *Progress toward Opaque*

Typedefs for C++0X
- ^ Doc No. 1898 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1898.pdf) : Michel Michaud, Michael Wong (6 October 2004) *Forwarding and inherited constructors*
- ^ Doc No. 1919 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1919.pdf) : Bjarne Stroustrup, Gabriel Dos Reis (11 December 2005) *Initializer lists*
- ^ Doc No. 1968 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf) : V Samko; J Willcock, J Järvi, D Gregor, A Lumsdaine (26 February 2006) *Lambda expressions and closures for C++*
- ^ Doc No. 1986 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1986.pdf) : Herb Sutter, Francis Glassborow (6 April 2006) *Delegating Constructors (revision 3)*
- ^ Doc No. 2016 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2016.html) : Hans Boehm, Nick Maclaren (21 April 2002) *Should volatile Acquire Atomicity and Thread Visibility Semantics?*
- ^ Doc No. 2142 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2142.html) : ISO/IEC DTR 19768 (12 January 2007) *State of C++ Evolution (between Portland and Oxford 2007 Meetings)*
- ^ Doc No. 2228 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2228.html) : ISO/IEC DTR 19768 (3 May 2007) *State of C++ Evolution (Oxford 2007 Meetings)*
- ^ Doc No. 2258 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2258.pdf) : G. Dos Reis and B. Stroustrup *Templates Aliases*
- ^ Doc No. 2280 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2280.html) : Lawrence Crowl (2 May 2007) *Thread-Local Storage*
- ^ Doc No. 2291 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2291.html) : ISO/IEC DTR 19768 (25 June 2007) *State of C++ Evolution (Toronto 2007 Meetings)*
- ^ Doc No. 2336 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2336.html) : ISO/IEC DTR 19768 (29 July 2007) *State of C++ Evolution (Toronto 2007 Meetings)*
- ^ Doc No. 2389 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2389.html) : ISO/IEC DTR 19768 (7 August 2007) *State of C++ Evolution (pre-Kona 2007 Meetings)*
- ^ Doc No. 2431 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2431.pdf) : SC22/WG21/N2431 = J16/07-0301 (2 October 2007), *A name for the null pointer: nullptr*
- ^ Doc No. 2432 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2432.html) : ISO/IEC DTR 19768 (23 October 2007) *State of C++ Evolution (post-Kona 2007 Meeting)*
- ^ Doc No. 2437 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2437.pdf) : Lois Goldthwaite (5 October 2007) *Explicit Conversion Operators*
- ^ Doc No. 2461 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2461.pdf) : ISO/IEC DTR 19768 (22 October 2007) *Working Draft, Standard for programming Language C++*
- ^ Doc No. 2507 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2507.html) : ISO/IEC DTR 19768 (4 February 2008) *State of C++ Evolution (pre-Bellevue 2008 Meeting)*
- ^ Doc No. 2544 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers

/2008/n2544.pdf) : Alan Talbot, Lois Goldthwaite, Lawrence Crowl, Jens Maurer (29 February 2008) *Unrestricted unions*
- ^ Doc No. 2565 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2565.html) : ISO/IEC DTR 19768 (7 March 2008) *State of C++ Evolution (post-Bellevue 2008 Meeting)*
- ^ Doc No. 2597 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2597.html) : ISO/IEC DTR 19768 (29 April 2008) *State of C++ Evolution (pre-Antipolis 2008 Meeting)*
- ^ Doc No. 2606 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2606.pdf) : ISO/IEC DTR 19768 (19 May 2008) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 2697 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2697.html) : ISO/IEC DTR 19768 (15 June 2008) *Minutes of WG21 Meeting 8–15 June 2008*
- ^ Doc No. 2798 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf) : ISO/IEC DTR 19768 (4 October 2008) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 2857 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2857.pdf) : ISO/IEC DTR 19768 (23 March 2009) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 2869 (http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2869.html) : ISO/IEC DTR 19768 (28 April 2009) *State of C++ Evolution (post-San Francisco 2008 Meeting)*
- ^ Doc No. 3000 (http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n3000.pdf) : ISO/ISC DTR 19769 (9 November 2009) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3014 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n3014.pdf) : Stephen D. Clamage (4 November 2009) *AGENDA, PL22.16 Meeting No. 53, WG21 Meeting No. 48, 8–13 March 2010, Pittsburgh, PA*
- ^ Doc No. 3082 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3082.pdf) : Herb Sutter (13 March 2010) *C++0x Meeting Schedule*
- ^ Doc No. 3092 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf) : ISO/ISC DTR 19769 (26 March 2010) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3126 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf) : ISO/ISC DTR 19769 (21 August 2010) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3225 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3225.pdf) : ISO/ISC DTR 19769 (27 November 2010) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3242 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf) : ISO/ISC DTR 19769 (28 February 2011) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3291: ISO/ISC DTR 19769 (5 April 2011) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3290: ISO/ISC DTR 19769 (5 April 2011) *FDIS, Standard for Programming Language C++*

Articles

- ^ Bjarne Stroustrup (2 January 2006). "A Brief Look at C++0x"

(http://www.artima.com/cppsource/cpp0x.html) . The C++ Source. http://www.artima.com/cppsource/cpp0x.html. Retrieved 23 March 2009.
- ^ Bjarne Stroustrup (1 May 2005). "The Design of C++0x: Reinforcing C++'s proven strengths, while moving into the future" (http://www.research.att.com/~bs/rules.pdf) . C/C++ Users Journal. http://www.research.att.com/~bs/rules.pdf. Retrieved 23 March 2009.
- ^ Raffaele Rialdi (16 September 2005). "Il futuro di C++ raccontato da Herb Sutter" (http://blogs.ugidotnet.org/raffaele/archive/2005/09/16/26570.aspx) . Web Log di Raffaele Rialdi. http://blogs.ugidotnet.org/raffaele/archive/2005/09/16/26570.aspx. Retrieved 23 March 2009.
- ^ Danny Kalev (21 July 2006). "The Explicit Conversion Operators Proposal" (http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=297) . InformIT. http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=297. Retrieved 23 March 2009.
- ^ Danny Kalev (11 July 2008). "Lambda Expressions and Closures" (http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=254) . InformIT. http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=254. Retrieved 23 March 2009.
- ^ Pete Becker (11 April 2006). "Regular Expressions" (http://www.ddj.com/cpp/185300414) . Dr. Dobb's Portal. http://www.ddj.com/cpp/185300414. Retrieved 23 March 2009.
- ^ Danny Kalev (10 March 2006). "The Type Traits Library" (http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=276) . InformIT. http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=276. Retrieved 23 March 2009.
- ^ Pete Becker (1 May 2005). "C++ Function Objects in TR: Getting from TR1 back to the Standard Library" (http://www.ddj.com/cpp/184401949) . Dr. Dobb's Portal. http://www.ddj.com/cpp/184401949. Retrieved 23 March 2009.
- ^ Howard E. Hinnant, Bjarne Stroustrup, and Bronek Kozicki (10 March 2008). "A Brief Introduction to Rvalue References" (http://www.artima.com/cppsource/rvalue.html) . The C++ Source. http://www.artima.com/cppsource/rvalue.html. Retrieved 23 March 2009.
- ^ "C++0x: The Dawning of a New Standard" (http://www.devx.com/SpecialReports/Door/38865) . DevX. 18 August 2008. http://www.devx.com/SpecialReports/Door/38865. Retrieved 23 March 2009.
- ^ "Static code analysis and the new language standard C++0x" (http://software.intel.com/en-us/articles/static-code-analysis-and-the-new-language-standard-c0x/) . Intel Software Network. 15 April 2010. http://software.intel.com/en-us/articles/static-code-analysis-and-the-new-language-standard-c0x/.
- ^ Bjarne Stroustrup (August 2009). "No 'Concepts' in C++0x" (http://accu.org/index.php/journals/1576) . accu.org. http://accu.org/index.php/journals/1576. Retrieved 29 June 2010.
- ^ "Explicating the new C++ standard (C++0x), and its implementation in VC10" (http://www.codeproject.com/KB/cpp/cpp10.aspx) . CodeProject.com. 8 April 2010. http://www.codeproject.com/KB/cpp/cpp10.aspx. Retrieved 13 February 2011.

External links

- **The C++ Standards** Committee (http://www.open-std.org/jtc1/sc22/wg21/)
- Bjarne Stroustrup's homepage (http://www.research.att.com/~bs/)
- Boost C++ Libraries (http://www.boost.org/)
- 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++0x (http://herbsutter.wordpress.com/)
- Anthony Williams' blog coverage of C++0x (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)

Categories: C++ | Computer standards | Articles with example C++ code

---