C++11

维基百科，自由的百科全书

C++11，先前被称作C++0x，即ISO/IEC 14882:2011，是目前的C++编程语言的正式标准。它取代第二版标准ISO/IEC 14882:2003(第一版ISO/IEC 14882:1998公开于1998年，第二版于2003年更新，分别通称C++98以及C++03，两者差异很小)。新的标准包含核心语言的新机能，而且扩展C++标准程序库，并入了大部分的C++ Technical Report 1程序库(数学的特殊函数除外)。最新的消息被公开在 ISO C++ 委员会网站(英文) (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/)。

ISO / IEC JTC1/SC22/WG21 C++ 标准委员会计划在2010年8月之前完成对最终委员会草案的投票，以及于2011年3月召开的标准会议完成国际标准的最终草案。然而，WG21预期ISO将要花费六个月到一年的时间才能正式发布新的 C++ 标准。为了能够如期完成，委员会决定致力于直至2006年为止的提案，忽略新的提案[1]。最终于2011年8月12日公布，并于2011年9月出版。

2012年2月28日的国际标准草案(N3376 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf))是最接近于现行标准的草案，差异仅有编辑上的修正。

像C++这样的编程语言，通过一种演化的的过程来发展其定义。这个过程不可避免地将引发与现有代码的兼容问题，在C++的发展过程中偶尔会发生。不过根据Bjarne Stroustrup(C++的创始人并且是委员会的一员)表示，新的标准将几乎100%兼容于现有标准。

目录

候选变更

C++的修订包含核心语言以及标准程序库。

在发展新标准的每个机能上，委员会采取了几个方向：

- 维持与C++98，可能的话还有C之间的稳定性与兼容性；
- 尽可能不通过核心语言的扩展，而是通过标准程序库来引进新的特色；
- 能够演进编程技术的变更优先；
- 改进 C++ 以帮助系统以及库设计，而不是引进只针对特别应用的新特色；
- 增进类型安全，提供对现行不安全的技术更安全的替代方案；
- 增进直接对硬件工作的能力与表现；
- 提供现实世界中问题的适当解决方案；
- 实行“zero-overhead”原则(某些功能要求的额外支持只有在该功能被使用时才能使用)；
- 使C++易于教授与学习

对初学者的注重被认为是重要的，因为他们构成了计算机程序员的主体。也因为许多初学者不愿扩展他们对 C++ 的知识，只限于使用他们对 C++ 专精的部分。此外，考虑到 C++ 被广泛的使用(包含应用领域和编程风格)，即便是最有经验的程序员在面对新的编程范式时也会成为初学者。

C++核心语言的扩充

C++委员会的主要焦点是在语言核心的发展上。核心语言将被大幅改善的领域包括多线程(或称为“多线程”)支持、泛型编程、统一的初始化，以及性能表现的加强。

在此分成4个区块来讨论核心语言的特色以及变更： 运行期表现强化、构造期表现强化、可用性强化，还有新的功能。某些特色可能会同时属于多个区块，但在此仅于其最具代表性的区块描述该特色。

核心语言的运行期表现强化

以下的语言机能主要用来提升某些性能表现，像是内存或是速度上的表现。

右值引用和 move 语义

在 C++03及之前的标准，临时对象(称为右值"R-values"，位于赋值运算符之右)无法被改变，在 C 中亦同(且被视为无法和 const T& 做出区分)。尽管在某些情况下临时对象的确会被改变，甚至也被视为是一个有用的漏洞。

C++11 增加一个新的非常数引用(reference)类型，称作右值引用(R-value reference)，标记为T &&。右值引用所引用的临时对象可以在该临时对象被初始化之后做修改，这是为了允许 move 语义。

C++03 性能上被长期被诟病的其中之一，就是其耗时且不必要的深度拷贝。深度拷贝会发生在当对象是以传值的方式传递。举例而言，std::vector<T> 是内部保存了 C-style 数组的一个包装，如果一个std::vector<T>的临时对象被构造或是从函数回返，

要将其存储只能通过生成新的std::vector<T>并且把该临时对象所有的数据复制进去。该临时对象和其拥有的内存会被摧毁。(为了讨论上的方便，这里忽略回返值优化)

在 C++11，一个std::vector的 "move 构造函数" 对某个vector的右值引用可以单纯地从右值复制其内部 C-style 数组的指针到新的 vector，然后留下空的右值。这个操作不需要数组的复制，而且空的临时对象的析构也不会摧毁内存。传回vector临时对象的函数不需要显式地传回std::vector<T>&&。如果vector没有 move 构造函数，那么复制构造函数将被调用，以const std::vector<T> &的正常形式。 如果它确实有 move 构造函数，那么就会调用 move 构造函数，这能够免除大幅的内存配置。

基于安全的理由，具名的参数将永远不被认定为右值，即使它是被如此声明的；为了获得右值必须使用 std::move<T>()。

```
bool is_r_value(int &&) { return true; }
bool is_r_value(const int &) { return false; }

void test(int && i)
{
    is_r_value(i); // i 為具名變數，即使被宣告成右值也不會被認定是右值。
    is_r_value(std::move<int&>(i)); // 使用 std::move<T>() 取得右值。
}
```

由于右值引用的用语特性以及对于左值引用(L-value references;regular references)的某些用语修正，右值引用允许开发者提供完美转发 (perfect function forwarding)。当与变长参数模板结合，这项能力允许函数模板能够完美地转送引用给其他接受这些特定引用的函数。最大的用处在于转送构造函数参数，创造出能够自动为这些特定引用调用正确构造函数的工厂函数(factory function)。

泛化的常数表示式

C++ 本来就已具备常数表示式(constant expression)的概念。像是 3+4 总是会产生相同的结果并且没有任何的副作用。常数表示式对编译器来说是优化的机会，编译器时常在编译期运行它们并且将值存入程序中。同样地，在许多场合下，C++ 规格要求使用常数表示式。例如在数组大小的定义上，以及枚举值(enumerator values)都要求必须是常数表示式。

然而，常数表示式总是在遇上了函数调用或是对象构造函数时就终结。所以像是以下的例子是不合法的：

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

int some_value[GetFive() + 5]; // 欲產生 10 個整數的陣列。 不合法的 C++ 寫法
```

这不是合法的 C++，因为 GetFive() + 5 并不是常数表示式。编译器无从得知 GetFive 实际上在运行期是常数。理论上而言，这个函数可能会影响全局参数，或者调用其他的非运行期(non-runtime)常数函数等。

C++11引进关键字 constexpr 允许用户保证函数或是对象构造函数是编译期常数。以上的例子可以被写成像是下面这样：

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

int some_value[GetFive() + 5]; // 欲產生 10 個整數的陣列。合法的C++11寫法
```

这使得编译器能够了解并去验证 GetFive 是个编译期常数。

对函数使用 constexpr 在函数可以做的事上面加上了非常严格的条件。首先，该函数的回返值类型不能为 void。第二点，函数的内容必须依照 "return expr" 的形式。第三点，在引用取代后，expr 必须是个常数表示式。这些常数表示式只能够调用其他被定义为 constexpr 的函数，或是其他常数表示式的数据参数。 最后一点，有着这样标签的函数直到在该编译单元内被定义之前是不能够被调用的。

参数也可以被定义为常数表示式值：

```
constexpr double forceOfGravity = 9.8;
constexpr double moonGravity = forceOfGravity / 6.0;
```

常数表示式的数据参数是隐式的常数。他们可以只存储常数表示式或常数表示式构造函数的结果。

为了从用户自定类型(user-defined type)构造常数表示式的数据参数，构造函数也可以被声明成 constexpr。与常数表示式函数一样，常数表示式的构造函数必须在该编译单元内使用之前被定义。他必须有着空的函数本体。它必须用常数表示式初始化他的成员(member)。而这种类型的析构函数应当是无意义的(trivial)，什么事都不做。

复制 constexpr 构造起来的类型也应该被定义为 constexpr，这样可以让他们从常数表示式的函数以值传回。类型的任何成员函数，像是复制构造函数、重载的运算符等等，只要他们符合常数表示式函数的定义，都可以被声明成 constexpr。这使得编译器能够在编译期进行类型的复制、对他们施行运算等等。

常数表示式函数或构造函数，可以以非常数表示式(non-constexpr)参数调用。就如同 constexpr 整数字面值能够指派给 non-constexpr 参数，constexpr 函数也可以接受 non-constexpr 参数，其结果存储于 non-constexpr 参数。constexpr 关键字

只有当表示式的成员都是 constexpr，才允许编译期常数性的可能。

对POD定义的修正

在标准C++，一个结构(struct)为了能够被当成 POD，必须遵守几条规则。有很好的理由使我们想让大量的类型符合这种定义，符合这种定义的类型能够允许产生与C兼容的对象布局(object layout)。然而，C++03的规则太严苛了。

C++11将会放宽关于POD的定义。

当class/struct是极简的(trivial)、属于标准布局(standard-layout)，以及他的所有非静态(non-static)成员都是POD时，会被视为POD。

一个极简的类型或结构符合以下定义：

1. 极简的默认构造函数。这可以使用默认构造函数语法，例如SomeConstructor() = default;
2. 极简的复制构造函数，可使用默认语法(default syntax)
3. 极简的赋值运算符，可使用默认语法(default syntax)
4. 极简的析构函数，不可以是虚拟的(virtual)

一个标准布局(standard-layout)的类型或结构符合以下定义：

1. 只有非静态的(non-static)数据成员，且这些成员也是符合标准布局的类型
2. 对所有non-static成员有相同的访问控制(public, private, protected)
3. 没有虚函数
4. 没有虚拟基类
5. 只有符合标准布局的基类
6. 没有和第一个定义的non-static成员相同类型的基类
7. 若非没有带有non-static成员的基类，就是最底层(继承最末位)的类型没有non-static数据成员而且至多一个带有non-static成员的基类。基本上，在该类型的继承体系中只会有一个类型带有non-static成员。

核心语言构造期表现的加强

外部模板

在标准C++中，只要在编译单元内遇到被完整定义的模板，编译器都必须将其实例化(instantiate)。这会大大增加编译时间，特别是模板在许多编译单元内使用相同的参数实例化。看起来没有办法告诉C++不要引发模板的实例化。

C++11将会引入外部模板这一概念。C++已经有了强制编译器在特定位置开始实例化的语法：

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

而C++所缺乏的是阻止编译器在某个编译单元内实例化模板的能力。C++11将简单地扩充前文语法如下：

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

这样就告诉编译器不要在该编译单元内将该模板实例化。

核心语言使用性的加强

这些特色存在的主要目的是为了使C++能够更容易使用。 举凡可以增进类型安全，减少代码重复，不易误用代码之类的。

初始化列表

标准C++从C带来了初始化列表(initializer list)的概念。这个构想是结构或是数组能够依据成员在该结构内定义的顺序通过给予的一串引用来产生。这些初始化列表是递归的，所以结构的数组或是包含其他结构的结构可以使用它们。这对静态列表或是仅是把结构初始化为某值而言相当有用。C++有构造函数，能够重复对象的初始化。但单单只有那样并不足以取代这项特色的所有机能。在C++03中，只允许在严格遵守POD的定义和限制条件的结构及类型上使用这项机能，非POD的类型不能使用，就连相当有用的STL容器std::vector也不行。

C++11将会把初始化列表的概念绑到类型上，称作std::initializer_list。这允许构造函数或其他函数像参数般地使用初始化列表。举例来说：

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

这将允许SequenceClass由一连串的整数构造，就像：

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

这个构造函数是种特殊的构造函数，称作初始化列表构造函数。有着这种构造函数的类型在统一初始化的时候会被特别对待。

类型std::initializer_list<>是个第一级的C++11标准程序库类型。然而他们只能够经由C++11编译器通过{}语法的使用被静态地构造 。这个列表一经构造便可复制，虽然这只是copy-by-reference。初始化列表是常数；一旦被创建，其成员均不能被改变，成员中的数据也不能够被变动。

因为初始化列表是真实类型，除了类型构造函数之外还能够被用在其他地方。正规的函数能够使用初始化列表作为引用。例如：

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

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

标准容器也能够以这种方式初始化：

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

统一的初始化

标准 C++ 在初始化类型方面有着许多问题。初始化类型有数种方法，而且交换使用时不会都产生相同结果。传统的构造函数语法，看起来像是函数声明，而且为了能使编译器不会弄错必须采取一些步骤。只有集合体和 POD 类型能够被集合式的初始化(使用 SomeType var = {/*stuff*/};).

C++11 将会提供一种统一的语法初始化任意的对象，它扩充了初始化列表语法：

```
struct BasicStruct
{
 int x;
 float y;
};

struct AltStruct
{
  AltStruct(int _x, float _y) : x(_x), y(_y) {}

private:
  int x;
  float y;
};

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

var1 的初始化的运作就如同 C-style 的初始化列表。每个公开的参数将被对应于初始化列表的值给初始化。隐式类型转换会在需要的时候被使用，这里的隐式类型转换不会产生范围缩限 (narrowing)。要是不能够转换，编译便会失败。(范围缩限 (narrowing)：转换后的类型无法表示原类型。如将 32-bit 的整数转换为 16-bit 或 8-bit 整数，或是浮点数转换为整数。) var2 的初始化则是简单地调用构造函数。

统一的初始化构造能够免除具体指定特定类型的必要：

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

IdString var3{"SomeName", 4};
```

该语法将会使用 const char * 参数初始化 std::string 。你也可以做像下面的事：

```
IdString GetString()
{
  return {"SomeName", 4}; // 注意這裡不需要明確的型別
}
```

统一初始化不会取代构造函数语法。仍然会有需要用到构造函数语法的时候。如果一个类型拥有初始化列表构造函数(TypeName(initializer_list<SomeType>);)，而初始化列表符合 sequence 构造函数的类型，那么它比其他形式的构造函数的优先权都来的高。C++11 版本的 std::vector 将会有初始化列表构造函数。这表示：

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

这将会调用初始化列表构造函数，而不是调用std::vector只接受一个尺寸参数产生相应尺寸 vector 的构造函数。要使用这个构造函数，用户必须直接使用标准的构造函数语法。

类型推导

在标准 C++(和 C)，使用参数必须明确的指出其类型。然而，随着模版类型的出现以及模板元编程的技巧，某物的类型，特别是函数定义明确的回返类型，就不容易表示。在这样的情况下，将中间结果存储于参数是件困难的事，可能会需要知道特定的元编程程序库的内部情况。

C++11 提供两种方法缓解上述所遇到的困难。首先，有被明确初始化的参数可以使用 auto 关键字。这会依据该初始化子(initializer)的具体类型产生参数：

```
auto someStrangeCallableType = boost::bind(&SomeFunction, _2, _1, someObject);
auto otherVariable = 5;
```

someStrangeCallableType 的类型就是模板函数 boost::bind 对特定引用所回返的类型。作为编译器语义分析责任的一部份，这个类型能够简单地被编译器决定，但用户要通过查看来判断类型就不是那么容易的一件事了。

otherVariable 的类型同样也是定义明确的，但用户很容易就能判别。它是个 int(整数)，就和整数字面值的类型一样。

除此之外，decltype 能够被用来在编译期决定一个表示式的类型。举例：

```
int someInt;
decltype(someInt) otherIntegerVariable = 5;
```

decltype 和 auto 一起使用会更为有用，因为 auto 参数的类型只有编译器知道。然而 decltype 对于那些大量运用运算符重载和特化的类型的代码的表示也非常有用。

auto 对于减少冗赘的代码也很有用。举例而言，程序员不用写像下面这样：

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

而可以用更简短的

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

这项差异随着程序员开始嵌套容器而更为显著，虽然在这种情况下 typedef 是一个减少代码的好方法。

decltype 所表示的类型可以和 auto 推导出来的不同。

```
#include <vector>

int main()
{
  const std::vector<int> v(1);
  auto a = v[0]; // a 為 int 型別
  decltype (v[0]) b = 0;   // b 為 const int& 型別，即
                           // std::vector<int>::operator[] (size_type) const 的回返型別
  auto c = 0;              // c 為 int 型別
  auto d = c;              // d 為 int 型別
  decltype(c) e;           // e 為 int 型別，c 實體的型別
  decltype((c)) f = e; // f 為 int& 型別，因為 (c) 是左值
  decltype(0) g;           // g為int型別，因為0是右值
}
```

以范围为基础的 for 循环

Boost C++ 定义了许多"范围 (range) "的概念。范围表现有如受控制的列表 (list)，持有容器中的两点。有序容器是范围概念的超集 (superset)，有序容器中的两个迭代器 (iterator) 也能定义一个范围。这些概念以及操作的算法，将被并入 C++11 标准程序库。不过 C++11 将会以语言层次的支持来提供范围概念的效用。

for 述句将允许简单的范围迭代：

```
int my_array[5] = {1, 2, 3, 4, 5};
for (int &x : my_array)
{
  x *= 2;
}
```

上面 for 述句的第一部份定义被用来做范围迭代的参数，就像被声明在一般 for 循环的参数一样，其作用域仅只于循环的范围。而在":"之后的第二区块，代表将被迭代的范围。这样一来，就有了能够允许 C-style 数组被转换成范围概念的概念图。这可以是 std::vector，或是其他符合范围概念的对象。

Lambda函数与表示式

在标准 C++，特别是当使用 C++ 标准程序库算法函数诸如 sort 和 find，用户经常希望能够在算法函数调用的附近定义一个临时的述部函数(又称谓词函数，predicate function)。由于语言本身允许在函数内部定义类型，可以考虑使用函数对象，然而这通常既麻烦又冗赘，也阻碍了代码的流程。此外，标准 C++ 不允许定义于函数内部的类型被用于模板，所以前述的作法是不可行的。

C++11 对 lambda 的支持可以解决上述问题。

一个 lambda 函数可以用如下的方式定义:

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

这个不具名函数的回返类型是 decltype(x+y)。只有在 lambda 函数符合"return expression"的形式下，它的回返类型才能被忽略。在前述的情况下，lambda 函数仅能为一个述句。

在一个更为复杂的例子中，回返类型可以被明确的指定如下：

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

本例中，一个临时的参数 z 被创建用来存储中间结果。如同一般的函数，z 的值不会保留到下一次该不具名函数再次被调用时。

如果 lambda 函数没有传回值(例如 void)，其回返类型可被完全忽略。

定义在与 lambda 函数相同作用域的参数引用也可以被使用。这种的参数集合一般被称作 closure (闭包)。

```
[]  // 沒有定義任何變數。使用未定義變數會導致錯誤。
[x, &y] // x 以傳值方式傳入(預設)，y 以傳參考方式傳入。
[&]   // 任何被使用到的外部變數皆隱式地以參考方式加以引用。
[=]   // 任何被使用到的外部變數皆隱式地以傳值方式加以引用。
[&, x]   // x 顯示地以傳值方式加以引用。其餘變數以參考方式加以引用。
[=, &z]   // z 顯示地以參考方式加以引用。其餘變數以傳值方式加以引用。
```

closure 被定义与使用如下:

```
std::vector<int> someList;
int total = 0;
std::for_each(someList.begin(), someList.end(), [&total](int x) {
  total += x;
});
std::cout << total;
```

上例可计算 someList 元素的总和并将其印出。 参数 total 是 lambda 函数 closure 的一部分，同时它以引用方式被传递入谓词函数， 因此它的值可被 lambda 函数改变。

若不使用引用的符号&，则代表参数以传值的方式传入 lambda 函数。 让用户可以用这种表示法明确区分参数传递的方法：传值，或是传引用。 由于 lambda 函数可以不在被声明的地方就地使用(如置入 std::function 对象中)； 这种情况下，若参数是以传引用的方式链接到 closure 中，是无意义甚至是危险的行为。

若 lambda 函数只在定义的作用域使用， 则可以用 [&] 声明 lambda 函数， 代表所有引用到 stack 中的参数，都是以引用的方式传入， 不必一一显式指明：

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

参数传入 lambda 函数的方式可能随实现有所变化，一般期望的方法是 lambda 函数能保留其作用域函数的 stack 指针，借此访问区域参数。

若使用 [=] 而非 [&]，则代表所有的引用的参数都是传值使用。

对于不同的参数，传值或传引用可以混和使用。 比方说，用户可以让所有的参数都以传引用的方式使用，但带有一个传值使用的参数：

```
int total = 0;
int value = 5;
[&, value](int x) { total += (x * value); };
```

total 是传引用的方式传入 lambda 函数，而 value 则是传值。

若一个 lambda 函数被定义于某类型的成员函数中，则可以使用该类型对象的引用，并且能够访问其内部的成员。

```
[](SomeType *typePtr) { typePtr->SomePrivateMemberFunction(); };
```

这只有当该 lambda 函数创建的作用域是在 SomeType 的成员函数内部时才能运作。

在成员函数中指涉对象的 this 指针，必须要显式的传入 lambda 函数， 否则成员函数中的 lambda 函数无法使用任何该对象的参数或函数。

```
[this]() { this->SomePrivateMemberFunction(); };
```

若是 lambda 函数使用 [&] 或是 [=] 的形式，this在 lambda 函数即为可见。

lambda 函数是编译器从属类型的函数对象； 这种类型名称只有编译器自己能够使用。如果用户希望将 lambda 函数作为参数传入，该类型必须是模版类型，或是必须创建一个 std::function 去获取 lambda 的值。使用 auto 关键字让我们能够存储 lambda 函数：

```
auto myLambdaFunc = [this]() { this->SomePrivateMemberFunction(); };
auto myOnheapLambdaFunc = new auto([=] { /*...*/ });
```

另一种的函数语法

标准C 函数声明语法对于C语言已经足够。 演化自 C 的 C++ 除了 C 的基础语法外，又扩充额外的语法。 然而，当 C++ 变得更为复杂时，它暴露出许多语法上的限制， 特别是针对函数模板的声明。 下面的示例，不是合法的 C++03：

```
template< typename LHS, typename RHS>
  Ret AddingFunc(const LHS &lhs, const RHS &rhs) {return lhs + rhs;} //Ret的型别必须是(lhs+rhs)的型别
```

Ret 的类型由 LHS与RHS相加之后的结果的类型来决定。 即使使用 C++11 新加入的 decltype 来声明 AddingFunc 的回返类型，依然不可行。

```
template< typename LHS, typename RHS>
  decltype(lhs+rhs) AddingFunc(const LHS &lhs, const RHS &rhs) {return lhs + rhs;} //不合法的 C++11
```

不合法的原因在于lhs 及 rhs 在定义前就出现了。 直到剖析器解析到函数原型的后半部，lhs 与 rhs 才是有意义的。

针对此问题，C++11 引进一种新的函数定义与声明的语法：

```
template< typename LHS, typename RHS>
  auto AddingFunc(const LHS &lhs, const RHS &rhs) -> decltype(lhs+rhs) {return lhs + rhs;}
```

这种语法也能套用到一般的函数定义与声明：

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

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

关键字 auto 的使用与其在自动类型推导代表不同的意义。

对象构造的改良

在标准C++中，构造函数不能调用其它的构造函数；每个构造函数必须自己初始化所有的成员或是调用一个共用的成员函数。基类的构造函数不能够直接作为派生类的构造函数；就算基类的构造函数已经足够，每个派生的类型仍必须实现自己的构造函数。类型中non-constant的数据成员不能够在声明的地方被初始化，它们只能在构造函数中被初始化。 C++11将会提供这些问题的解决方案。

C++11允许构造函数调用其他构造函数，这种做法称作委托或转接(delegation)。 仅仅只需要加入少量的代码，就能让数个构造函数之间达成功能复用(reuse)。 Java以及C#都有提供这种功能。C++11 语法如下：

```
class SomeType {
  int number;
  string name;
  SomeType( int i, string& s ) : number(i), name(s){}
public:
  SomeType( )           : SomeType( 0, "invalid" ){}
  SomeType( int i )     : SomeType( i, "guest" ){}
  SomeType( string& s ) : SomeType( 1, s ){ PostInit(); }
};
```

C++03中，构造函数运行退出代表对象构造完成； 而允许使用转接构造函数的 C++11 则是以"任何"一个构造函数退出代表构造完成。 使用转接的构造函数，函数本体中的代码将于被转接的构造函数完成后继续运行(如上例的 PostInit())。 若基底类型使用了转接构造函数，则派生类的构造函数会在"所有"基底类型的构造函数都完成后， 才会开始运行。

C++11 允许派生类手动继承基底类型的构造函数， 编译器可以使用基底类型的构造函数完成派生类的构造。 而将基类的构造函数带入派生类的动作， 无法选择性地部分带入， 要不就是继承基类全部的构造函数，要不就是一个都不继承(不手动带入)。 此外，若牵涉到多重继承，从多个基底类型继承而来的构造函数不可以有相同的函数签名(signature)。 而派生类的新加入的构造函数也不可以和继承而来的基底构造函数有相同的函数签名，因为这相当于重复声明。

语法如下：

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

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

此语法等同于 DerivedClass 声明一个DerivedClass(int) 的构造函数。 同时也因为 DerivedClass 有了一个继承而来的构造函数，所以不会有默认构造函数。

另一方面，C++11可以使用以下的语法完成成员初始化：

```
class SomeClass
{
public:
  SomeClass() {}
  explicit SomeClass(int iNewValue) : iValue(iNewValue) {}

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

若是构造函数中没有设置iValue的初始值，则会采用类定义中的成员初始化，令iValue初值为5。在上例中，无参数版本的构造函数，iValue便采用默认所定义的值； 而带有一个整数参数的构造函数则会以指定的值完成初始化。

成员初始化除了上例中的赋值形式(使用"=")外，也可以采用构造函数以及统一形的初始化(uniform initialization，使用"{}")。

显式虚函数重载

在 C++ 里，在子类中容易意外的重载虚函数。举例来说:

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

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

Derived::some_func 的真实意图为何？程序员真的试图重载该虚函数，或这只是意外？这也可能是 base 的维护者在其中加入了一个与 Derived::some_func 同名且拥有相同签名的虚函数。

另一个可能的状况是，当基类中的虚函数的签名被改变，子类中拥有旧签名的函数就不再重载该虚函数。因此，如果程序员忘记修改所有子类，运行期将不会正确调用到该虚函数正确的实现。

C++11 将加入支持用来防止上述情形产生，并在编译期而非运行期捕获此类错误。为保持向后兼容，此功能将是选择性的。其语法如下：

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

struct Derived : Base {
    virtual void some_func(int) override;   // 錯誤格式: Derive::some_func 並沒有 override Base::some_func
    virtual void some_func(float) override; // OK: 顯式改寫
};
```

编译器会检查基底类型是否存在一虚拟函数，与派生类中带有声明override 的虚拟函数，有相同的函数签名(signature)；若不存在，则会回报错误。

C++11 也提供指示字final，用来避免类型被继承，或是基底类型的函数被改写：

```
struct Base1 final { };

struct Derived1 : Base1 { }; // 錯誤格式: class Base1 以標明為 final

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

struct Derived2 : Base2 {
    void f(); // 錯誤格式: Base2::f 以標明為 final
};
```

以上的示例中，virtual void f() final;声明一新的虚拟函数，同时也表明禁止派生函数改写原虚拟函数。

override与final都不是语言关键字(keyword)，只有在特定的位置才有特别含意，其他地方仍旧可以作为一般指示字(identifier)使用。

空指针

早在 1972 年，C语言诞生的初期，常数 0 带有常数及空指针的双重身分。 C 使用 preprocessor macro NULL 表示空指针， 让 NULL 及 0 分别代表空指针及常数 0。 NULL 可被定义为 ((void*)0) 或是 0。

C++ 并不采用 C 的规则，不允许将 void* 隐式转换为其他类型的指针。 为了使代码 char* c = NULL; 能通过编译，NULL 只能定义为 0。 这样的决定使得函数重载无法区分代码的语义：

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

C++ 建议 NULL 应当定义为 0，所以foo(NULL); 将会调用 foo(int)， 这并不是程序员想要的行为，也违反了代码的直观性。0 的歧义在此处造成困扰。

C++11 引入了新的关键字来代表空指针常数：nullptr，将空指针和整数 0 的概念拆开。 nullptr 的类型为nullptr_t，能隐式转换为任何指针或是成员指针的类型，也能和它们进行相等或不等的比较。 而nullptr不能隐式转换为整数，也不能和整数做比较。

为了向下兼容，0 仍可代表空指针常数。

```
char* pc = nullptr;     // OK
int * pi = nullptr;     // OK
int    i = nullptr;     // error

foo(pc);           // 呼叫 foo(char *)
```

强类型枚举

在标准C++中，枚举类型不是类型安全的。枚举类型被视为整数，这使得两种不同的枚举类型之间可以进行比较。C++03 唯一提供的安全机制是一个整数或一个枚举型值不能隐式转换到另一个枚举别型。 此外，枚举所使用整数类型及其大小都由实现方

法定义，皆无法明确指定。 最后，枚举的名称全数暴露于一般范围中，因此两个不同的枚举，不可以有相同的枚举名。（好比 enum Side{ Right, Left }; 和 enum Thing{ Wrong, Right }; 不能一起使用。）

C++11 引进了一种特别的 "枚举类"，可以避免上述的问题。使用 enum class 的语法来声明：

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

此种枚举为类型安全的。枚举类型不能隐式地转换为整数；也无法与整数数值做比较。（表示式 Enumeration::Val4 == 101 会触发编译期错误）。

枚举类型所使用类型必须显式指定。在上面的示例中，使用的是默认类型 int，但也可以指定其他类型：

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

枚举类型的语汇范围(scoping)定义于枚举类型的名称范围中。 使用枚举类型的枚举名时，必须明确指定其所属范围。 由前述枚举类型 Enum2 为例，Enum2::Val1是有意义的表示法， 而单独的 Val1 则否。

此外，C++11 允许为传统的枚举指定使用类型：

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

枚举名 Val1 定义于 Enum3 的枚举范围中(Enum3::Val1)，但为了兼容性， Val1 仍然可以于一般的范围中单独使用。

在 C++11 中，枚举类型的前置声明 (forward declaration) 也是可行的，只要使用可指定类型的新式枚举即可。 之前的 C++ 无法写出枚举的前置声明，是由于无法确定枚举参数所占的空间大小， C++11 解决了这个问题：

```
```

```
enum Enum1;                     // 不合法的 C++ 與 C++11; 無法判別大小
enum Enum2 : unsigned int;      // 合法的 C++11
enum class Enum3;               // 合法的 C++11，列舉類別使用預設型別 int
enum class Enum4: unsigned int; // 合法的 C++11
enum Enum2 : unsigned short;    // 不合法的 C++11，Enum2 已被聲明為 unsigned int
```

角括号

标准 C++ 的剖析器一律将 ">>" 视为右移运算符。 但在样板定义式中，绝大多数的场合其实都代表两个连续右角括号。 为了避免剖析器误判，撰码时不能把右角括号连着写。

C++11 变更了剖析器的解读规则；当遇到连续的右角括号时，优先解析右角括号为样板引用的退出符号。 如果解读过程中出现普通括号("(" 与 ")")，这条规则产生变化：

```
template<bool bTest> SomeType;
std::vector<SomeType<1>2>> x1;   // 解讀為 std::vector of "SomeType<true> 2>",
                                 // 非法的表示式， 整數 1 被轉換為 bool 型別 true
std::vector<SomeType<(1>2)>> x1; // 解讀為 std::vector of "SomeType<false>",
                                 // 合法的 C++11 表示式， (1>2) 被轉換為 bool 型別 false
```

显式类型转换子

C++ 为了避免用户自定的单引用构造函数被当成隐式类型转换子，引入了关键字 explicit 修饰字。 但是，在编译器对对象调用隐式类型转换的部分，则没有任何着墨。 比方说，一个 smart pointer 类型具有一个operator bool()， 被定义成若该 smart pointer 保管任何资源或指针，则传回 true，反之传回 false。 遇到这样的代码时：if(smart_ptr_variable)，编译器可以借由 operator bool() 隐式转换成布尔值， 和测试原生指针的方法一样。 但是这类隐式转换同样也会发生在非预期之处。由于 C++ 的 bool 类型也是算数类型，能隐式换为整数甚至是浮点数。 拿对象转换出的布尔值做布尔运算以外的数学运算，往往不是程序员想要的。

在 C++11 中，关键字 explicit 修饰符也能套用到类型转换子上。如同构造函数一样，它能避免类型转换子被隐式转换调用。但 C++11 特别针对布尔值转换提出规范，在 if 条件式，循环，逻辑运算等需要布尔值的地方，编译器能为符合规范的表示式调用用户自定的布尔类型转换子。

模板的别名

在进入这个主题之前，各位应该先弄清楚“模板”和“类型”本质上的不同。class template (类型模板，是模板)是用来产生 template class (模板类型，是类型)。在标准 C++，typedef 可定义模板类型一个新的类型名称，但是不能够使用 typedef

来定义模板的别名。举例来说：

```
template< typename first, typename second, int third>
class SomeType;

template< typename second>
typedef SomeType<OtherType, second, 5> TypedefName; // 在C++是不合法的
```

这不能够通过编译。

为了定义模板的别名，C++11 将会增加以下的语法：

```
template< typename first, typename second, int third>
class SomeType;

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

using 也能在 C++11 中定义一般类型的别名，等同 typedef：

```
typedef void (*PFD)(double);            // 傳統語法
using PFD = void (*)(double);           // 新增語法
```

无限制的unions

在标准 C++ 中，并非任意的类型都能做为 union 的成员。比方说，带有 non-trivial 构造函数的类型就不能是 union 的成员。在新的标准里，移除了所有对 union 的使用限制，除了其成员仍然不能是引用类型。 这一改变使得 union 更强大，更有用，也易于使用。[1]

以下为 C++11 中 union 使用的简单样例：

```
struct point
{
  point() {}
  point(int x, int y): x_(x), y_(y) {}
  int x_, y_;
};
union
{
  int z;
  double w;
  point p;  // 不合法的 C++; point 有一 non-trivial 建構式
```

```
        // 合法的 C++11
};
```

这一改变仅放宽 union 的使用限制，不会影响既有的旧代码。

核心语言能力的提升

这些机能提供了C++语言能够做一些事情是以前所不能达成的，或是在以前需要繁琐的写法、要求一些不可移植的程序库。

变长参数模板

在 C++11 之前，不论是类模板或是函数模板，都只能按其被声明时所指定的样子，接受一组固定数目的模板参数 ； C++11 加入新的表示法，允许任意个数、任意类别的模板参数，不必在定义时将参数的个数固定。

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

模板类 tuple 的对象，能接受不限个数的 typename 作为它的模板形参：

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

实参的个数也可以是 0，所以 class tuple<> someInstanceName 这样的定义也是可以的。

若不希望产生实参个数为 0 的变长参数模板，则可以采用以下的定义：

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

变长参数模板也能运用到模板函数上。 传统 C 中的 printf 函数，虽然也能达成不定个数的形参的调用，但其并非类别安全。 以下的样例中，C++11 除了能定义类别安全的变长参数函数外，还能让类似 printf 的函数能自然地处理非内置类别的对象。 除了在模板参数中能使用...表示不定长模板参数外，函数参数也使用同样的表示法代表不定长参数。

```

```

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

其中，Params 与 parameters 分别代表模板与函数的变长参数集合， 称之为参数包(parameter pack)。参数包必须要和运算符"..."搭配使用，避免语法上的歧义。

变长参数模板中，变长参数包无法如同一般参数在类或函数中使用； 因此典型的手法是以递归的方法取出可用参数，参看以下的 C++11 printf 样例：

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

template<typename T, typename... Args>
void printf(const char* s, T value, Args... args)
{
  while (*s)
  {
    if (*s == '%' && *(++s) != '%')
    {
      std::cout << value;
      printf(*s ? ++s : s, args...); // 即便当 *s == 0 也会产生调用，以检测更多的类型参数。
      return;
    }
    std::cout << *s++;
  }
  throw std::logic_error("extra arguments provided to printf");
}
```

printf 会不断地递归调用自身：函数参数包 args... 在调用时， 会被模板类别匹配分离为 T value和 Args... args。 直到 args... 变为空参数，则会与简单的 printf(const char *s) 形成匹配，退出递归。

另一个例子为计算模板参数的个数，这里使用相似的技巧展开模板参数包 Args...：

```
template<>
struct count<> {
    static const int value = 0;
};

template<typename T, typename... Args>
struct count<T, Args...> {
    static const int value = 1 + count<Args...>::value;
};
```

虽然没有一个简洁的机制能够对变长参数模板中的值进行迭代，但使用运算符"..."还能在代码各处对参数包施以更复杂的展开操作。举例来说，一个模板类的定义：

```
template <typename... BaseClasses> class ClassName : public BaseClasses...
{
public:

   ClassName (BaseClasses&&... baseClasses) : BaseClasses(baseClasses)... {}
}
```

BaseClasses... 会被展开成类型 ClassName 的基底类； ClassName 的构造函数需要所有基类的右值引用，而每一个基类都是以传入的参数做初始化(BaseClasses(baseClasses)...)。

在函数模板中，变长参数可以和右值引用搭配，达成形参的完美转送 (perfect forwarding)：

```
template<typename TypeToConstruct> struct SharedPtrAllocator
{
  template<typename... Args> std::shared_ptr<TypeToConstruct> ConstructWithSharedPtr(Args&&... params)
  {
    return tr1::shared_ptr<TypeToConstruct>(new TypeToConstruct(std::forward<Args>(params)...));
  }
}
```

参数包 parms 可展开为 TypeToConstruct 构造函数的形参。 表达式std::forward<Args>(params) 可将形参的类别信息保留(利用右值引用)，传入构造函数。 而运算符"..."则能将前述的表达式套用到每一个参数包中的参数。这种工厂函数(factory function)的手法， 使用 std::shared_ptr 管理配置对象的内存，避免了不当使用所产生的内存泄漏(memory leaks)。

此外，变长参数的数量可以藉以下的语法得知：

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

SomeStruct<Type1, Type2>::size 是 2，而 SomeStruct<>::size 会是 0。 (sizeof...(Args) 的结果是编译期常数。)

新的字符串字面值

标准C++提供了两种字符串字面值。第一种，包含有双引号，产生以空字符结尾的const char数组。第二种有着前标L，产生以空字符结尾的const wchar_t数组，其中wchar_t代表宽字符。对于Unicode编码的支持尚付阙如。

为了加强C++编译器对Unicode的支持，类别char的定义被修改为其大小至少能够存储UTF-8的8位编码，并且能够容纳编译器的基本字符集的任何成员。

C++11 将支持三种Unicode编码方式：UTF-8，UTF-16，和UTF-32。除了上述char定义的变更， C++11将增加两种新的字符类别：char16_t和char32_t。它们各自被设计用来存储UTF-16 以及UTF-32的字符。

以下展示如何产生使用这些编码的字符串字面值：

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

第一个字符串的类别是通常的const char[]；第二个字符串的类别是const char16_t[]；第三个字符串的类别是const char32_t[]。

当创建Unicode字符串字面值时，可以直接在字符串内插入Unicode codepoints。C++11提供了以下的语法：

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

在'\u'之后的是16个比特的十六进制数值；它不需要'0x'的前标。识别字'\u'代表了一个16位的Unicode codepoint；如果要输入32位的codepoint，使用'\U'和32个比特的十六进制数值。只有有效的Unicode codepoints能够被输入。举例而言，codepoints在范围U+D800—U+DFFF之间是被禁止的，它们被保留给UTF-16编码的surrogate pairs。

有时候避免手动将字符串换码也是很有用的，特别是在使用XML文件或是一些脚本语言的字面值的时候。 C++11将提供raw(未加工的)字符串字面值：

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

在第一个例子中，任何包含在( )括号(标准已经从[]改为())当中的都是字符串的一部分。其中"和\字符不需要经过转义(escaped)。在第二个例子中，"delimiter(开始字符串，只有在遇到)delimiter"才代表退出。其中delimiter可以是任意的字符串，能够允许用户在未加工的字符串字面值中使用)字符。 未加工的字符串字面值能够和宽字面值或是Unicode字面值结合：

```
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.)"
```

用户自定义的字面值

标准C++提供了数种字面值。字符"12.5"是能够被编译器解释为数值12.5的double类别字面值。然而，加上"f"的后置，像是"12.5f"，则会产生数值为12.5的float类别字面值。在C++规范中字面值的后置是固定的，而且C++代码并不允许创立新的字面后置。

C++11 开放用户定义新的字面修饰符(literal modifier)，利用自定义的修饰符完成由字面值构造对象。

字面值转换可以区分为两个阶段：转换前与转换后 (raw 与 cooked)。 转换前的字面值指特定字符串行，而转换后的字面值则代表另一种类别。 如字面值1234，转换前的字面值代表 '1', '2', '3', '4' 的字符串行； 而转换后，字面值代表整数值1234。另外，字面值0xA转换前是串行'0', 'x', 'A'；转换后代表整数值 10。

多任务内存模型

C++标准委员会计划统一对多线程编程的支持。

这将涉及两个部分：第一、设计一个可以使多个线程在一个进程中共存的内存模型；第二、为线程之间的交互提供支持。第二部分将由程序库提供支持，更多请看线程支持。

在多个线程可能会访问相同内存的情形下，由一个内存模型对它们进行调度是非常有必要的。遵守模型规则的程序是被保证正确运行的，但违反规则的程序会发生不可预料的行为，这些行为依赖于编译器的优化和内存一致性的问题。

thread-local的存储期限

在多线程环境下，让各线程拥有各自的参数是很普遍的。这已经存在于函数的区域参数，但是对于全局和静态参数都还不行。

新的thread_local存储期限(在现行的static、dynamic和automatic之外)被作为下个标准而提出。线程区域的存储期限会借由存储指定字thread_local来表明。

static对象(生命周期为整个程序的运行期间)的存储期限可以被thread-local给替代。就如同其他使用static存储期的参数，thread-local对象能够以构造函数初始化并以析构函数摧毁。

使用或禁用对象的默认函数

在传统C++中，若用户没有提供， 则编译器会自动为对象生成默认构造函数(default constructor)、 复制构造函数(copy constructor)，赋值运算符(copy assignment operator operator=) 以及析构函数(destructor)。另外，C++也为所有的类定义了数个全局运算符(如operator delete及operator new)。当用户有需要时，也可以提供自定义的版本改写上述的函数。

问题在于原先的c++无法精确地控制这些默认函数的生成。 比方说，要让类型不能被拷贝，必须将复制构造函数与赋值运算符声明为private，并不去定义它们。 尝试使用这些未定义的函数会导致编译期或链接期的错误。 但这种手法并不是一个理想的解决方案。

此外，编译器产生的默认构造函数与用户定义的构造函数无法同时存在。 若用户定义了任何构造函数，编译器便不会生成默认构造函数； 但有时同时带有上述两者提供的构造函数也是很有用的。 目前并没有显式指定编译器产生默认构造函数的方法。

C++11 允许显式地表明采用或拒用编译器提供的内置函数。例如要求类型带有默认构造函数，可以用以下的语法：

```
struct SomeType
{
  SomeType() = default; // 預設建構式的顯式聲明
  SomeType(OtherType value);
};
```

另一方面，也可以禁止编译器自动产生某些函数。如下面的例子，类型不可复制：

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

禁止类型以operator new配置内存：

```
struct NonNewable
{
  void *operator new(std::size_t) = delete;
};
```

此种对象只能生成于 stack 中或是当作其他类型的成员，它无法直接配置于 heap 之中，除非使用了与平台相关，不可移植的手法。（使用 placement new 运算符虽然可以在用户自配置的内存上调用对象构造函数，但在此例中其他形式的 new 运算符一并被上述的定义 屏蔽("name hiding")，所以也不可行。）

= delete的声明(同时也是定义)也能适用于非内置函数， 禁止成员函数以特定的形参调用：

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

若尝试以 double 的形参调用 f()，将会引发编译期错误， 编译器不会自动将 double 形参转型为 int 再调用f()。 若要彻底的禁止以非int的形参调用f()，可以将= delete与模板相结合：

```
struct OnlyInt
{
  void f(int i);
  template<class T> void f(T) = delete;
};
```

long long int类别

在 32 比特系统上，一个 long long int 是保有至少 64 个有效比特的整数类别。C99 将这个类别引入了标准 C 中，目前大多数的 C++ 编译器也支持这种类别。C++11 将把这种类别添加到标准 C++ 中。

静态assertion

C++提供了两种方法测试assertion(声明)：宏assert以及预处理器指令#error。但是这两者对于模版来说都不合用。宏在运行期测试assertion，而预处理器指令则在前置处理时测试assertion，这时候模版还未能实例化。所以它们都不适合来测试牵扯到模板参数的相关特性。

新的机能会引进新的方式可以在编译期测试assertion，只要使用新的关键字static_assert。 声明采取以下的形式：

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

这里有一些如何使用static_assert的例子：

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

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

当常数表达式值为false时，编译器会产生相应的错误信息。第一个例子是预处理器指令#error的替代方案；第二个例子会在每个模板类型Check生成时检查assertion。

静态assertion在模板之外也是相当有用的。例如，某个算法的实现依赖于long long类别的大小比int还大，这是标准所不保证的。 这种假设在大多数的系统以及编译器上是有效的，但不是全部。

允许sizeof运算符作用在类型的数据成员上，无须明确的对象

在标准C++，sizeof可以作用在对象以及类别上。但是不能够做以下的事：

```
struct SomeType { OtherType member; };

sizeof(SomeType::member); // 直接由SomeType型别取得非静態成員的大小，C++03不行。 C++11允許
```

这会传回OtherType的大小。C++03并不允许这样做，所以会引发编译错误。C++11将会允许这种使用。

垃圾回收机制

是否会自动回收那些无法被使用到（unreachable）的动态分配对象由实现决定。

C++标准程序库的变更

C++11 标准程序库有数个新机能。其中许多可以在现行标准下实现，而另外一些则依赖于(或多或少)新的 C++11 核心语言机能。

新的程序库的大部分被定义于C++标准委员会的Library Technical Report（称TR1），于2005年发布。各式 TR1 的完全或部分实现目前提供在命名空间 std::tr1。C++11 会将其移置于命名空间 std 之下。

标准库组件上的升级

目前的标准库能受益于 C++11 新增的一些语言特性。举例来说，对于大部份的标准库容器而言，像是搬移内含大量元素的容器，或是容器之内对元素的搬移，基于右值引用（Rvalue reference）的 move 构造函数都能优化前述动作。在适当的情况下，标准库组件将可利用 C++11 的语言特性进行升级。这些语言特性包含但不局限以下所列:

- 右值引用和其相关的 move 支持
- 支持 UTF-16 编码，和 UTF-32 字符集
- 变长参数模板（与右值引用搭配可以达成完美转送（perfect forwarding））
- 编译期常数表达式
- Decltype
- 显式类别转换子
- 使用或禁用对象的默认函数

此外，自 C++ 标准化之后已经过许多年。现有许多代码利用到了标准库；这同时揭露了部份的标准库可以做些改良。其中之一是标准库的内存配置器（allocator)。C++11将会加入一个基于作用域模型的内存配置器来支持现有的模型。

线程支持

虽然 C++11 会在语言的定义上提供一个内存模型以支持线程，但线程的使用主要将以 C++11 标准库的方式呈现。

C++11 标准库会提供类型 thread（std::thread)。若要运行一个线程，可以创建一个类型 thread 的实体，其初始参数为一个函数对象，以及该函数对象所需要的参数。通过成员函数 std::thread::join() 对线程会合的支持，一个线程可以暂停直到其它线程运行完毕。若有底层平台支持，成员函数 std::thread::native_handle() 将可提供对原生线程对象运行平台特定的操作。

对于线程间的同步，标准库将会提供适当的互斥锁（像是 std::mutex，std::recursive_mutex 等等）和条件参数（std::condition_variable 和 std::condition_variable_any）。前述同步机制将会以 RAII 锁（std::lock_guard 和 std::unique_lock）和锁相关算法的方式呈现，以方便程序员使用。

对于要求高性能，或是极底层的工作，有时或甚至是必须的，我们希望线程间的通信能避免互斥锁使用上的开销。以原子操作来访问内存可以达成此目的。针对不同情况，我们可以通过显性的内存屏障改变该访问内存动作的可见性。

对于线程间异步的传输，C++11 标准库加入了 以及 std::packaged_task 用来包装一个会传回异步结果的函数调用。 因为缺少结合数个 future 的功能，和无法判定一组 promise 集合中的某一个 promise 是否完成，futures 此一提案因此而受到了批评。

更高级的线程支持，如线程池，已经决定留待在未来的 Technical Report 加入此类支持。更高级的线程支持不会是 C++11 的一部份，但设想是其最终实现将创建在目前已有的线程支持之上。

std::async 提供了一个简便方法以用来运行线程，并将线程绑定在 std::future。用户可以选择一个工作是要多个线程上异步的运行，或是在一个线程上运行并等待其所需要的数据。默认的情况，实现可以根据底层硬件选择前面两个选项的其中之一。另外在较简单的使用情形下，实现也可以利用线程池提供支持。

多元组类别

多元组是一个内由数个异质对象以特定顺序排列而成的数据结构。多元组可被视为是 struct 其数据成员的一般化。

由 TR1 演进而来的 C++11 多元组类别将受益于 C++11 某些特色像是变长参数模板。TR1 版本的多元组类别对所能容纳的对象个数会因实现而有所限制，且实现上需要用到大量的宏技巧。相反的，C++11 版本的多元组型基本上于对其能容纳的对象个数没有限制。然而，编译器对于模板实体化的递归深度上的限制仍旧影响了元组类别所能容纳的对象个数（这是无法避免的情况）；C++11 版本的多元组型不会把这个值让用户知道。

使用变长参数模板，多元组类别的声明可以长得像下面这样：

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

底下是一个多元组类别的定义和使用情况：

```
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);  // 将 proof 的第一個元素賦值給 lengthy (索引從零開始起跳)
std::get<3>(proof) = " Beautiful!";  // 修改 proof 的第四個元素
```

我们可以定义一个多元组类别对象 proof 而不指定其内容，前提是 proof 里的元素其类别定义了默认构造函数 (default constructor)。此外，以一个多元组类别对象赋值给另一个多元组类别对象是可能的，但只有在以下情况：若这两个多元组类别相同，则其内含的每一个元素其类别都要定义拷贝构造函数 (copy constructor)；否则的话，赋值操作符右边的多元组其内含元素的类别必须能转换成左边的多元组其对应的元素类别，又或者赋值操作符左边的多元组其内含元素的类别必须定义适当的构造函数。

```
typedef std::tuple< int , double, string       > tuple_1 t1;
typedef std::tuple< char, short , const char * > tuple_2 t2 ('X', 2, "Hola!");
t1 = t2 ;  // 可行。前兩個元素會作型別轉換，
           // 第三個字串元素可由 'const char *' 所建構。
```

多元组类型对象的比较运算是可行的(当它们拥有同样数量的元素)。此外，C++11 提供两个表达式用来检查多元组类型的一些特性 (仅在编译期做此检查)。

- std::tuple_size<T>::value 回传多元组 T 内的元素个数，
- std::tuple_element<I, T>::type 回传多元组 T 内的第 I 个元素的类别

散列表

在过去，不断有要求想将散列表(无序关系式容器)引进标准库。只因为时间上的限制，散列表才没有被标准库所采纳。虽然，散列表在最糟情况下(如果出现许多冲突(collision) 的话)在性能上比不过平衡树。但实际运用上，散列表的表现则较佳。

因为标准委员会还看不到有任何机会能将开放寻址法标准化，所以目前冲突仅能通过链地址法 (linear chaining) 的方式处理。为避免与第三方库发展的散列表发生名称上的冲突，前缀将采用 unordered 而非 hash。

库将引进四种散列表，其中差别在于底下两个特性：是否接受具相同键值的项目 (Equivalent keys)，以及是否会将键值映射到相对应的数据 (Associated values)。

散列表类型	有无关系值	接受相同键值
std::unordered_set	否	否
std::unordered_multiset	否	是
std::unordered_map	是	否
std::unordered_multimap	是	是

上述的类型将满足对一个容器类型的要求，同时也提供访问其中元素的成员函数：insert，erase，begin，end。

散列表不需要对现有核心语言做扩展(虽然散列表的实现会利用到 C++11 新的语言特性)，只会对头文件 <functional> 做些许扩展，并引入 <unordered_set> 和 <unordered_map> 两个头文件。对于其它现有的类型不会有任何修改。同时，散列表也不会依赖其它标准库的扩展功能。

正则表达式

过去许多或多或少标准化的程序库被创建用来处理正则表达式。有鉴于这些算法的使用非常普遍，因此标准程序库将会包含他们，并使用各种面向对象语言的潜力。

这个新的程序库，被定义于<regex>头文件，由几个新的类型所组成：

- 正则表达式(模式)以样板类 basic_regex 的实体表示
- 模式匹配的情况以样板类 match_results 的实体表示

函数 regex_search 是用来搜索模式；若要搜索并取代，则要使用函数 regex_replace，该函数会回传一个新的字符串。算法regex_search 和 regex_replace 接受一个正则表达式(模式)和一个字符串，并将该模式匹配的情况存储在 struct match_results。

底下描述了 match_results 的使用情况：

```
const char *reg_esp = "[ ,.\\t\\n;:]" ;  // 分隔字元列表

std::regex rgx(reg_esp) ;  // 'regex' 是樣板類 'basic_regex' 以型別為 'char'
                           //  的參數具現化的實體
std::cmatch match ;  // 'cmatch' 是樣板類 match_results' 以型別為 'const char *'
                     // '的參數具現化的實體
const char *target = "Polytechnic University of Turin " ;

// 辨別所有被分隔字元所分隔的字
if( regex_search( target, match, rgx ) )
{
  // 若此種字存在

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

```
    cout << str << "\n" ;
  }
}
```

注意双反斜线的使用，因为 C++ 将反斜线作为转义字符使用。但 C++11 的raw string可以用来避免此一问题。库 <regex> 不需要改动到现有的头文件，同时也不需要对现有的语言作扩展。

通用智能指针

这些指针是由 TR1 智能指针演变而来。注意！智能指针是类型而非一般指针。

shared_ptr 是一引用计数 (reference-counted) 指针，其行为与一般 C++ 指针即为相似。在 TR1 的实现中，缺少了一些一般指针所拥有的特色，像是别名或是指针运算。C++11新增前述特色。

一个 shared_ptr 只有在已经没有任何其它 shared_ptr 指向其原本所指向对象时，才会销毁该对象。

一个 weak_ptr 指向的是一个被 shared_ptr 所指向的对象。该 weak_ptr 可以用来决定该对象是否已被销毁。weak_ptr 不能被解引用；想要访问其内部所保存的指针，只能通过 shared_ptr。有两种方法可达成此目的。第一，类型 shared_ptr 有一个以 weak_ptr 为参数的构造函数。第二，类型 weak_ptr 有一个名为 lock 的成员函数，其回返值为一个 shared_ptr。weak_ptr 并不拥有它所指向的对象，因此不影响该对象的销毁与否。

底下是一个 shared_ptr 的使用样例：

```
int main( )
{
    std::shared_ptr<double> p_first(new double) ;

    {
        std::shared_ptr<double> p_copy = p_first ;

        *p_copy = 21.2;

    }  // 此時 'p_copy' 會被銷毀，但動態分配的 double 不會被銷毀。

    return 0;  // 此時 'p_first' 會被銷毀，動態分配的 double 也會被銷毀 (因為不再有指針指向它)。
}
```

auto_ptr 将会被 C++ 标准所废弃，取而代之的是 unique_ptr。 unique_ptr 提供 auto_ptr 大部份特性，唯一的例外是 auto_ptr 的不安全、隐性的左值搬移。不像 auto_ptr，unique_ptr 可以存放在 C++11 提出的那些能察觉搬移动作的容器之中。

可扩展的随机数功能

C 标准库允许使用rand函数来生成伪随机数。不过其算法则取决于各程序库开发者。C++ 直接从 C 继承了这部份，但是 C++11 将会提供产生伪乱数的新方法。

C++11 的随机数功能分为两部分： 第一，一个乱数生成引擎，其中包含该生成引擎的状态，用来产生乱数。第二，一个分布，这可以用来决定产生乱数的范围，也可以决定以何种分布方式产生乱数。乱数生成对象即是由乱数生成引擎和分布所构成。

不同于 C 标准库的 rand；针对产生乱数的机制，C++11 将会提供三种算法，每一种算法都有其强项和弱项：

样板类	整数/浮点数	质量	速度	状态数*
linear_congruential	整数	低	中等	1
subtract_with_carry	两者皆可	中等	快	25
mersenne_twister	整数	佳	快	624

C++11 将会提供一些标准分布：uniform_int_distribution（离散型均匀分布），bernoulli_distribution（伯努利分布），geometric_distribution（几何分布），poisson_distribution（卜瓦松分布），binomial_distribution（二项分布），uniform_real_distribution（离散型均匀分布）， exponential_distribution（指数分布），normal_distribution（正态分布）和 gamma_distribution（伽玛分布）。

底下描述一个乱数生成对象如何由乱数生成引擎和分布构成：

```
std::uniform_int_distribution<int> distribution(0, 99); // 以離散型均勻分佈方式產生 int 亂數，範圍落在 0 到
std::mt19937 engine; // 建立亂數生成引擎
auto generator = std::bind(distribution, engine); // 利用 bind 將亂數生成引擎和分布組合成一個亂數生成物件
int random = generator();  // 產生亂數
```

包装引用

我们可以通过实体化样板类 reference_wrapper 得到一个包装引用（wrapper reference）。包装引用类似于一般的引用。对于任意对象，我们可以通过模板类 ref 得到一个包装引用（至于 constant reference 则可通过 cref 得到）。

当样板函数需要形参的引用而非其拷贝，这时包装引用就能派上用场：

```
// 此函數將得到形參 'r' 的引用並對 r 加一
void f (int &r)  { r++; }
```

```
// 樣板函式
template<class F, class P> void g (F f, P t)  { f(t); }

int main()
{
    int i = 0 ;
    g (f, i) ;  // 實體化 'g<void (int &r), int>'
                // 'i' 不會被修改
    std::cout << i << std::endl;  // 輸出 0

    g (f, std::ref(i));  // 實體化 'g<void(int &r),reference_wrapper<int>>'
                         // 'i' 會被修改
    std::cout << i << std::endl;  // 輸出 1
}
```

这项功能将加入头文件 <utility> 之中，而非通过扩展语言来得到这项功能。

多态函数对象包装器

针对函数对象的多态包装器(又称多态函数对象包装器)在语义和语法上和函数指针相似，但不像函数指针那么狭隘。只要能被调用，且其参数能与包装器兼容的都能以多态函数对象包装器称之(函数指针，成员函数指针或仿函数)。

通过以下例子，我们可以了解多态函数对象包装器的特性:

```
std::function<int (int, int)> func;  // 利用樣板類 'function'
                                      // 建立包裝器
std::plus<int> add;  // 'plus' 被宣告為 'template<class T> T plus( T, T ) ;'
                     //  因此 'add' 的型別是 'int add( int x, int y )'
func = &add;  // 可行。'add' 的型參和回返值型別與 'func' 相符

int a = func (1, 2);  // 注意: 若包裝器 'func' 沒有參考到任何函式
                      // 會丟出 'std::bad_function_call' 例外

std::function<bool (short, short)> func2 ;
if(!func2) { // 因為尚未賦值與 'func2' 任何函式，此條件式為真

    bool adjacent(long x, long y);
    func2 = &adjacent ;  // 可行。'adjacent' 的型參和回返值型別可透過型別轉換進而與 'func2' 相符

    struct Test {
        bool operator()(short x, short y);
    };
    Test car;
    func = std::ref(car);  // 樣板類 'std::ref' 回傳一個 struct 'car'
                           // 其成員函式 'operator()' 的包裝
}
func = func2;  // 可行。'func2' 的型參和回返值型別可透過型別轉換進而與 'func' 相符
```

模板类 function 将定义在头文件 <functional>，而不须更动到语言本身。

用于元编程的类别属性

对于那些能自行创建或修改本身或其它程序的程序，我们称之为元编程。这种行为可以发生在编译或运行期。C++ 标准委员会已经决定引进一组由模板实现的库，程序员可利用此一库于编译期进行元编程。

底下是一个以元编程来计算指数的例子：

```
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;
```

许多算法能作用在不同的数据类别；C++ 模板支持泛型，这使得代码能更紧凑和有用。然而，算法经常会需要目前作用的数据类别的信息。这种信息可以通过类别属性（type traits）于模板实体化时将该信息萃取出来。

类别属性能识别一个对象的种类和有关一个类别（class）（或 struct）的特征。头文件 <type_traits> 描述了我们能识别那些特征。

底下的例子说明了模板函数‘elaborate’是如何根据给定的数据类别，从而实体化某一特定的算法（algorithm.do_it）。

```
// 演算法一
template< bool B > struct Algorithm {
    template<class T1, class T2> int do_it (T1 &, T2 &)  { /*...*/ }
};

// 演算法二
template<> struct Algorithm<true> {
    template<class T1, class T2> int do_it (T1, T2)  { /*...*/ }
};

// 根據給定的型別，實體化之後的 'elaborate' 會選擇演算法一或二
template<class T1, class T2>
int elaborate (T1 A, T2 B)
{
    // 若 T1 為 int 且 T2 為 float，選用演算法二
    // 其它情況選用演算法一
    return Algorithm<std::is_integral<T1>::value && std::is_floating_point<T2>::value>::do_it( A, B ) ;
}
```

通过定义在 <type_transform> 的类别属性，自定的类别转换是可能的（在模板中，static_cast 和 const_cast 无法适用所有情况）。

此种编程技巧能写出优美、简洁的代码；然而除错是此种编程技巧的弱处：编译期的错误信息让人不知所云，运行期的除错更是困难。

用于计算函数对象回返类型的统一方法

要在编译期决定一个样板仿函数的回返值类别并不容易，特别是当回返值依赖于函数的参数时。举例来说：

```
struct Clear {
    int    operator()(int);     // 參數與回返值的型別相同
    double operator()(double);  // 參數與回返值的型別相同
};

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

实体化样板类 Calculus<Clear>，Calculus 的仿函数其回返值总是和 Clear 的仿函数其回返值具有相同的类别。然而，若给定类型 Confused:

```
struct Confused {
    double operator()(int);     // 參數與回返值的型別不相同
    int    operator()(double);  // 參數與回返值的型別不相同
};
```

企图实体化样板类 Calculus<Confused> 将导致 Calculus 的仿函数其回返值和类型 Confused 的仿函数其回返值有不同的类别。对于 int 和 double 之间的转换，编译器将给出警告。

模板 std::result_of 被TR1 引进且被 C++11 所采纳，可允许我们决定和使用一个仿函数其回返值的类别。底下，CalculusVer2 对象使用 std::result_of 对象来推导其仿函数的回返值类别：

```
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;
};
```

如此一来，在实体化 CalculusVer2<Confused> 其仿函数时，不会有类别转换，警告或是错误发生。

模板 std::result_of 在 TR1 和 C++11 有一点不同。TR1 的版本允许实现在特殊情况下，可以无法决定一个函数调用其回返值类别。然而，因为 C++11支持了decltype，实现被要求在所有情况下，皆能计算出回返值类别。

已被移除或是不包含在 C++11 标准的特色

预计由 Technical Report 提供支持:

- 模块 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2316.pdf)
- 十进制类别
- 数学专用函数

延后讨论:

- Concepts (概念 (C++))
- 更完整或必备的垃圾回收支持
- Reflection
- Macro Scopes

被移除或废弃的特色

- 循序点 (sequence point)，这个术语正被更为易懂的描述所取代。一个运算可以发生 (is sequenced before) 在另一个运算之前；又或者两个运算彼此之间没有顺序关系 (are unsequenced)。
- export
- exception specifications
- std::auto_ptr 被 std::unique_ptr 取代。
- 仿函数基类别 (std::unary_function, std::binary_function)、函数指针适配器、类型成员指针适配器以及绑定器 (binder)。

编译器实现

C++编译器对C++11新特性的支持情况：

- Visual C++ 2010 : C++0x Core Language Features In VC10: The Table

(http://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx)
- Visual C++ 2010与Visual C++ 2012支持的C++11特性的对比列表：C++11 Features (Modern C++) (http://msdn.microsoft.com/en-us/library/vstudio/hh567368.aspx)
- GCC 4.8.1已实现C++11标准的所有主要语言特性: Status of Experimental C++11 Support in GCC 4.8 (http://gcc.gnu.org/gcc-4.8/cxx0x_status.html)

关系项目

- C++ Technical Report 1
- C11，C 编程语言的最新标准
- C++14，计划中的 C++ 标准

参考资料

1. ^ N2544 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2544.pdf)

C++标准委员会文件

- ^ Doc No. 1401 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1401.pdf): Jan Kristoffersen (2002年10月21日) Atomic operations with multi-threaded environments
- ^ Doc No. 1402 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1402.html): Doug Gregor (2002年10月22日) 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 (2002年11月8日) 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 (2003年3月3日) 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 (2003年3月3日) 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 (2003年4月7日) 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 (2003年3月27日) 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 (April 10, 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 (April 9, 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 (April 9, 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 (April 9, 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 (April 18, 2003) Reflective Metaprogramming in C++
- ^ Doc No. 1676 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1676.htm): Bronek Kozicki (September 9, 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 (September 10, 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 (September 12, 2004) Decltype (and auto)
- ^ Doc No. 1717 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1717.pdf): Francis Glassborow, Lois Goldthwaite (November 5, 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 (October 21, 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 (October 20, 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 (January 14, 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 (April 29, 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 (May 2, 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 (August 29, 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 (June 24, 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 (June 24, 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 (October 20, 2005) Specifying C++ concepts
- ^ Doc No. 1891 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1891.pdf): Walter E. Brown (October 18, 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 (October 6, 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 (December 11, 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 (February 26, 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 (April 6, 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 (April 21, 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 (January 12, 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 (May 3, 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 (May 2, 2007) Thread-Local Storage
- ^ Doc No. 2291 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2291.html): ISO/IEC DTR 19768 (June 25, 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 (July 29, 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 (August 7, 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 (October 2, 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 (October 23, 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 (October 5, 2007) Explicit Conversion Operators
- ^ Doc No. 2461 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2461.pdf): ISO/IEC DTR 19768 (October 22, 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 (February 4, 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 (February 29, 2008) Unrestricted unions
- ^ Doc No. 2565 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2565.html): ISO/IEC DTR 19768 (March 7, 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 (April 29, 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

(May 19, 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 (June 15, 2008) Minutes of WG21 Meeting June 8–15, 2008
- ^ Doc No. 2798 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf): ISO/IEC DTR 19768 (October 4, 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 (March 23, 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 (April 28, 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 (November 9, 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 (November 4, 2009) AGENDA, PL22.16 Meeting No. 53, WG21 Meeting No. 48, March 8–13, 2010, Pittsburgh, PA
- ^ Doc No. 3082 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3082.pdf): Herb Sutter (2010年3月13日) C++0x Meeting Schedule
- ^ Doc No. 3092 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf): ISO/ISC DTR 19769 (2010年3月26日) 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 (2010年8月21日) 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 (2010年11月27日) 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 (2011年2月28日) Working Draft, Standard for Programming Language C++
- ^ Doc No. 3290 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3290.pdf): ISO/ISC DTR 19769 (2011年4月11日) Working Draft, Standard for Programming Language C++

文章

- ^ a b The C++ Source Bjarne Stroustrup(2006年1月2日)A Brief Look at C++0x
- ^ C/C++ Users Journal Bjarne Stroustrup (May, 2005) The Design of

C++0x: Reinforcing C++'s proven strengths, while moving into the future
- Web Log di Raffaele Rialdi(2005年9月16日)Il futuro di C++ raccontato da Herb Sutter
- Informit.com(2006年8月5日)The Explicit Conversion Operators Proposal
- Informit.com(2006年7月25日)Introducing the Lambda Library
- Dr. Dobb's Portal Pete Becker(2006年4月11日)Regular Expressions TR1's regex implementation
- Informit.com(2006年7月25日)The Type Traits Library
- Dr. Dobb's Portal Pete Becker(2005年5月11日)C++ Function Objects in TR1
- The C++ Source Howard E. Hinnant, Bjarne Stroustrup, and Bronek Kozicki(2008年3月10日)A Brief Introduction to Rvalue References (http://www.artima.com/cppsource/rvalue.html)
- DevX.com Special Report(2008年8月18日)C++0x: The Dawning of a New Standard (http://www.devx.com/SpecialReports/Door/38865)

外部链接

- The C++ Standards Committee (http://www.open-std.org/jtc1/sc22/wg21/)
- Bjarne Stroustrup's homepage (http://www.research.att.com/~bs/)
- 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/)
- A talk on C++0x given by Bjarne Stroustrup at the University of Waterloo (http://www.csclub.uwaterloo.ca/media/C++0x%20-%20An%200verview.html)
- A quick and dirty introduction to C++0x (as of November 2007) (http://www.pvv.org/~oma/cpp0x_aquadi_nov_2007.pdf)
- The State of the Language: An Interview with Bjarne Stroustrup (August 15, 2008) (http://www.devx.com/SpecialReports/Article/38813/0/page/1)
- Working draft for the C++ language, October 4, 2008 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf)