C++11

维基百科，自由的百科全书

C++11，先前被稱作**C++0x**，即ISO/IEC 14882:2011，是C++程式語言的一个標準。它取代第二版標準ISO/IEC 14882:2003（第一版ISO/IEC 14882:1998公開於1998年，第二版於2003年更新，分别通稱C++98以及C++03，两者差异很小），且已被C++14取代。相比于C++03，C++11標準包含核心語言的新機能，而且擴展C++標準程式庫，併入了大部分的C++ Technical Report 1程式庫（數學的特殊函式除外）。 ISO／IEC JTC1/SC22/WG21 C++標準委員會計劃在2010年8月之前完成對最終委員會草案的投票，以及於2011年3月召開的標準會議完成國際標準的最終草案。然而，WG21預期ISO將要花費六個月到一年的時間才能正式發佈新的C++標準。為了能夠如期完成，委員會決定致力於直至2006年為止的提案，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日的國際標準草案[1]是最接近于C++11标准的草案，差异仅有编辑上的修正。

像C++這樣的程式語言，透過一種演化的的過程來發展其定義。這個過程不可避免地將引發與現有程式碼的相容問題，在C++的發展過程中偶爾會發生。不過根據比雅尼·斯特劳斯特鲁普（C++的創始人並且是委員會的一員）表示，新的標準將幾乎100%相容於現有標準。

目录

设计原则

C++的修訂包含核心語言以及標準程式庫。

在發展新標準的每個機能上，委員會採取了幾個方向：

- 維持穩定性和與C++98，可能的話還有C之間的兼容性；
- 儘可能不透過核心語言的擴展，而是透過標準程式庫來引進新的特色；
- 能夠演進編程技術的變更優先；
- 改進C++以幫助系統以及函式庫設計，而不是引進只針對特別應用的新特色；
- 增進型別安全，提供對現行不安全的技術更安全的替代方案；
- 增進直接對硬體工作的能力與表現；
- 提供現實世界中問題的適當解決方案；
- 實行“zero-overhead”原則（某些功能要求的額外支援只有在該功能被使用時才能使用）；
- 使C++易於教授與學習

关照初學者被認為是重要的，因為他們構成了計算機程序員的主體。也因為許多初學者不願擴展他們對C++的知識，只限於使用他們對C++專精的部分。此外，考慮到C++被廣泛的使用（包含應用領域和編程風格），即便是最有經驗的程序員在面對新的編程范式時也會成為初學者。

C++核心語言的擴充

C++委員會的主要作用之一是改善語言核心。核心語言將被大幅改善的領域包括多緒（或称为“多线程”）支援、泛型編程、統一的初始化，以及效能表現的加強。

在此分成4個區塊來討論核心語言的特色以及變更： 執行期表現強化、建構期表現強化、可用性強化，還有新的功能。某些特性可能會同時屬於多個區塊，但在此僅於其最具代表性的區塊描述。

核心語言的執行期表現強化

以下的語言機能主要用來提升某些效能表現，像是記憶體或是速度上的表現。

右值引用和 move語意

在C++03及之前的標準，臨時對象（稱為右值"R-values"，因为它们通常位於賦值運算子右側）無法被改變，在C中亦同（且被視為等同于**const T&**）。儘管如此，在某些情況下臨時對象仍然可能會被改變，但这种表现也被視為是一個有用的漏洞。

C++11增加一個新的非常數引用（reference）型別，稱作右值引用（R-value reference），標記為`T &&`。右值引用所绑定的臨時對象可以在該臨時對象被初始化之後做修改，這是為了允許move語意。

C++03低性能问题的之一，就是在以传值方式传递对象时隐式发生的耗時且不必要的深度拷貝。舉例而言，`std::vector<T>`本质上是一个C-style陣列及其大小的封裝，如果一個`std::vector<T>`的臨時物件是在函式内部或者函数返回时创建，要將其儲存就只能透過生成新的`std::vector<T>`並且把該臨時物件所有的資料複製过去（為了討論上的方便，這裡忽略返回值优化）。然后該臨時物件會被析构，其使用的記憶體会被释放。

在C++11，把一個`vector`的右值引用作为参数`std::vector`的"move建構子"，可以把右值参数所绑定的`vector`内部的指向C-style陣列的指標複製给新的`vector`，然後把该指標置null。由于临时变量不会被再次使用，所以不会有代码去访问该null指针；又因为该指针为null，当该临时对象超出作用域时曾经指向的内部C-style陣列

所使用的内存不会被释放。因此，该操作不仅无形中免去了深拷贝的开销，而且还很安全。

右值引用作为数据类型的引入，使得函数可以重载区分它的参数是值类型、传统的左值引用还是右值引用。这让除了标准库的现有代码无须任何改动就能等到性能提升。一个返回`std::vector<T>`的函数的返回类型无须为了调用move构造函数而显式修改为`std::vector<T>&&`，因为临时对象自动作为右值。（但是，如果`std::vector<T>`是没有move构造函数的C++03版，由于传统的左值引用也可以绑定到临时对象上，因此具有`const std::vector<T>&`参数的复制构造函数会被调用，导致一次显著的内存分配。）

出于安全的考虑，推行了一些限制。具名的变量被認定為左值，即使它是被宣告为右值引用数据类型；為了獲得右值必須使用显式类型转换，如模板函数`std::move<T>()`。右值引用所绑定的对象应该只在特定情境下被修改，主要用于move构造函数中。

```
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，即使被宣告成右值引用类型，i作為实参表达式也不會被認定是右值表达式。
    is_r_value(std::move<int&>(i)); // 使用std::move<T>()取得右值。
}
```

由於右值引用的语义特性以及對於左值引用（L-value references;regular references）的某些语义修正，右值引用让開發者能够提供函数参数的完美轉發（perfect function forwarding）。當與不定長參數模板結合，這項能力允許函式模板能夠完美地轉送参数給其他接受這些特定参数的函式。最大的用處是轉送建構子參數，創造出能夠自動為這些特定参数呼叫正確建構式的工廠函式（factory function）。这个用法可以在C++标准库中的emplace_back方法中看到。

泛化的常數表示式

C++本來就已具備常數表示式（constant expression）的概念。像是3+4總是會產生相同的結果並且沒有任何的副作用。常數表示式對編譯器來說是最佳化的機會，編譯器時常在編譯期執行它們並且將值存入程式中。同樣地，在許多場合下，C++規格要求使用常數表示式。例如在陣列大小的定義上，以及列舉值（enumerator values）都要求必須是常數表示式。

然而，常數表示式不能含有函式呼叫或是物件建構式。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
```

這在C++03中是不合法的，因為`GetFive() + 5`並不是常數表示式。C++03編譯器無從得知`GetFive`實際上在執行期是常數。理論上而言，這個函式可能會影響全域變數，或者呼叫其他的非執行期（non-runtime）常數函式等。

C++11引進關鍵字`constexpr`允許使用者保證函式或是物件建構式是編譯期常數。以上的例子可以被寫成像是下面這樣：

```
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
```

這使得編譯器能夠瞭解並去驗證`GetFive`是個編譯期常數。

用`constexpr`修饰函数将限制函式的行为。首先，該函式的回返值型別不能為void。第二，函式的內容必須依照"return *expr*"的形式。第三，在参数替换後，`expr`必須是個常數表示式。這些常數表示式只能夠呼叫其他被定義為`constexpr`的函式，或是其他常數表示式的資料變數。最後，有著這樣修饰符的函式直到在該編譯單元內被定義之前是不能夠被呼叫的。

声明为constexpr的函数也可以像其他函数一样用于常量表达式以外的地方，此时不需要满足后两点。

C++11之前，可以在常量表达式中使用的的变量必须被声明为const，用常量表达式来初始化，并且必须是整型或枚举类型。C++11去除了变量必须是整型或枚举类型的限制，只要变量使用了constexpr关键字来定义：

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

这些变量都是隐式常量，必须使用常量表达式来初始化。

為了让使用者自定义型别（user-defined type）参与建構常量表示式，建構式也可以用constexpr来声明。與constexpr函式一樣，constexpr建構式必須在該編譯單元內使用之前被定義。它的函数体必须为空。它必须用常量表示式初始化他的成員（member）。而這種型別的解構式應當是平凡的（trivial）。

拥有constexpr构造函数的类型的复制构造函数通常也應該被定義為constexpr，以便该类型的对象以值传递的方式从constexpr函数返回。该類別的任何成員函式，像是複製建構式、运算符重载函数等等，只要他們符合常數表达式函式的定義，都可以被宣告成constexpr。使得編譯器能夠在編譯期進行類別的複製、對他們施行運算等等。

常數表达式函式或建構式，可以以非常數表示式（non-constexpr）作为參數喚起。就如同constexpr整數字面值能夠指派給non-constexpr變數，constexpr函式也可以接受non-constexpr參數，其結果儲存於non-constexpr變數。constexpr關鍵字只有當表示式的成員都是constexpr，才允許編譯期常數性的可能。

對POD定義的修正

在C++03中，一個类（class）或結構（struct）要想被作为POD，必須遵守幾條規則。符合這種定義的型別能夠產生與C相容的物件內存佈局（object layout），而且可以被静态初始化。但C++03标准严格限制了何种类型与C兼容或可以被静态初始化的，尽管并不存在技术原因导致编译器无法处理。如果创建一个C++03 POD类型，然后为其添加一个非虚成员函数，这个类型就不再是POD类型了，从而无法被静态初始化，也不再与C兼容，尽管其内存布局并没有发生变化。

C++11通过把POD概念划分成两个概念：*平凡的（trivial）*和标准布局*（standard-layout）*，放寬了關於POD的定義。

一个平凡的类型可以被静态初始化，同时意味着使用memcpy来复制数据是合法的，而无须使用复制构造函数。平凡的类型对象的生命周期开始于其存储空间被分配时，而不是其构造函数完成时。

一個平凡的的類別或結構符合以下定義：

1. 平凡的預設建構式。這可以使用預設建構式語法，例如SomeConstructor() = default;
2. 平凡的複製建構式和move构造函数，可使用預設語法（default syntax）
3. 平凡的賦值運算子和move赋值操作符，可使用預設語法（default syntax）
4. 平凡的解構式，不可以是虛函数（virtual）

只有在类没有虚基类和虚成员函数时，构造函数才是平凡的。复制构造函数和赋值操作符还额外要求所有非静态数据成员都是平凡的。

一个符合标准布局的类封装成员的方式与C兼容。一個標準佈局（standard-layout）的類別或結構符合以下定義：

1. 只有非靜態的（non-static）資料成員，且這些成員也是符合標準佈局的型別
2. 對所有non-static成員有相同的存取控制（public，private，protected）
3. 沒有虛擬函式
4. 沒有虛擬基礎類別
5. 只有符合標準佈局的基礎類別
6. 沒有和第一個定義的non-static成員相同型別的基礎類別

7. 若非沒有帶有non-static成員的基礎類別，就是最底層（繼承最末位）的類別沒有non-static資料成員而且至多一個帶有non-static成員的基礎類別。基本上，在該類別的繼承體系中只會有一個類別帶有non-static成員。

一个类、结构、联合只有在其是平凡的、符合标准布局，并且所有非静态成员和基类都是POD时，才被视为POD。

通过划分，使得放弃一个特性而不失去另一个成为可能。一个具有复杂的复制和move构造函数的类可能不是平凡的，但是它可能符合标准布局，从而能与C程序交互。类似地，一个同时具有public和private数据成员的类不符合标准布局，但它可以是平凡的，从而能够使用`memcpy`来复制。

核心語言建構期表現的加強

外部模板

在標準C++中，只要在編譯單元內遇到被完整定義的模板，编译器都必須將其实例化（instantiate）。這會大大增加編譯時間，特別是模板在許多編譯單元內使用相同的參數具現化。看起来沒有辦法告訴C++不要引發模板的具現化。

C++11將會引入外部模板这一概念。C++已經有了強制編譯器在特定位置开始具現化的語法：

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

而C++所缺乏的是阻止編譯器在某個編譯單元內具現化模板的能力。C++11將簡單地擴充前文語法如下：

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

這样就告訴編譯器*不要*在該編譯單元內將該模板具現化。

使用时，如下例：

```
std::vector<MyClass> va;
```

核心語言使用性的加強

這些特色存在的主要目的是為了使C++能夠更容易使用。舉凡可以增進型別安全，減少程式碼重複，不易誤用程式碼之類的。

初始化列表

標準C++從C帶來了初始化列表（initializer list）的概念。這個構想是結構或是数组能夠依據成員在該結構內定義的順序透過給予的一串引數來產生。這些初始化列表是遞迴的，所以結構的数组或是包含其他結構的結構可以使用它們。這對靜態列表或是僅是把結構初始化為某值而言相當有用。C++有构造函数，能夠重複对象的初始化。但單單只有那樣並不足以取代這項特色的所有機能。在C++03中，只允許在嚴格遵守POD的定義和限制條件的結構及類別上使用這項機能，非POD的型別不能使用，就連相當有用的STL容器std::vector也不行。

C++11將會把初始化列表的概念綁到型別上，稱作`std::initializer_list`。這允許构造函数或其他函数像參數般地使用初始化列表。舉例來說：

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

這將允許SequenceClass由一連串的整數构造，就像：

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

這個构造函数是種特殊的构造函数，稱作初始化列表构造函数。有著這種构造函数的類別在統一初始化的時候會被特別對待。

類別std::initializer_list<>是個第一級的C++11標準程式庫型別。然而他們只能夠經由C++11編譯器透過{}語法的使用被靜態地构造。這個列表一經构造便可複製，雖然這只是copy-by-reference。初始化列表是常數；一旦被建立，其成員均不能被改變，成員中的資料也不能夠被變動。

因為初始化列表是真實型別，除了類別构造式之外還能夠被用在其他地方。正規的函数能夠使用初始化列表作為形参。例如：

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

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

標準容器也能夠以這種方式初始化：

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

統一的初始化

標準C++在初始化型別方面有著許多問題。初始化型別有數種方法，而且交換使用時不會都產生相同結果。傳統的建構式語法，看起來像是函式宣告，而且為了能使編譯器不會弄錯必須採取一些步驟。只有集合體和POD型別能夠被集合式的初始化（使用SomeType var = {/*stuff*/};）。

C++11將會提供一種統一的語法初始化任意的物件，它擴充了初始化串列語法：

```
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的初始化的運作就如同C-style的初始化串列。每個公開的變數將被對應於初始化串列的值給初始化。隱式型別轉換會在需要的時候被使用，這裡的隱式型別轉換不會產生範圍縮限（narrowing）。要是不能夠轉換，編譯便會失敗。（範圍縮限（narrowing）：轉換後的型別無法表示原型別。如將32-bit的整數轉換為16-bit或8-bit整數，或是浮點數轉換為整數。）var2的初始化則是簡單地呼叫建構式。

統一的初始化建構能夠免除具體指定特定型別的必要：

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

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

該語法將會使用const char *參數初始化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
}
```

統一初始化不會取代建構式語法。仍然會有需要用到建構式語法的時候。如果一個類別擁有初始化列表构造函数（TypeName(initializer_list<SomeType>);），而初始化串列与构造函数的参数类型一致，那麼它比其他形式的建構式的優先權都來的高。C++11版本的std::vector將會有初始化串列建構式。這表示：

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

這將會呼叫初始化串列建構式，而不是呼叫std::vector只接受一個尺寸參數產生相應尺寸vector的建構式。要使用這個建構式，使用者必須直接使用標準的建構式語法。

型別推導

在標準C++和C，使用變數必須明確的指出其型別。然而，隨著模版型別的出現以及模板超編程的技巧，某物的型別，特別是函式定義明確的回返型別，就不容易表示。在這樣的情況下，將中間結果儲存於變數是件困難的事，可能會需要知道特定的超編程程式庫的內部情況。

C++11提供兩種方法緩解上述所遇到的困難。首先，有被明確初始化的變數可以使用auto關鍵字。這會依據該初始化子（initializer）的具體型別產生變數：

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

someStrangeCallableType的型別就是模板函式boost::bind對特定引數所回返的型別。作為編譯器語意分析責任的一部份，這個型別能夠簡單地被編譯器決定，但使用者要透過檢視來判斷型別就不是那麼容易的一件事了。

otherVariable的型別同樣也是定義明確的，但使用者很容易就能判別。它是個int（整數），就和整數字面值的型別一樣。

除此之外，decltype能夠被用來在編譯期決定一個表示式的型別。舉例：

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

decltype和auto一起使用會更為有用，因為auto變數的型別只有編譯器知道。然而decltype對於那些大量運用運算子重載和特化的型別的程式碼的表示也非常有用。

auto對於減少冗贅的程式碼也很有用。舉例而言，程式員不用寫像下面這樣：

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

而可以用更簡短的

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

這項差異隨著程式員開始嵌套容器而更為顯著，雖然在這種情況下typedef是一個減少程式碼的好方法。

decltype所表示的型別可以和auto推導出來的不同。

```cpp
#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
}
```

基于范围的for迴圈

for语句將允許簡單的範圍迭代：

```cpp
int my_array[5] = {1, 2, 3, 4, 5};
// double the value of each element in my_array:
for (int &x : my_array)
{
  x *= 2;
}
// similar but also using type inference for array elements
for (auto &x : my_array) {
    x *= 2;
}
```

上面for述句的第一部份定義被用來做範圍迭代的变量，就像被宣告在一般for迴圈的变量一樣，其作用域僅只於迴圈的範圍。而在":"之後的第二區塊，代表將被迭代的範圍。这种for语句还可以用于C型数组，初始化列表，和任何定义了begin()和end()来返回首尾迭代器的类型。

Lambda函式與表示式

在標準C++，特別是當使用C++標準程式庫演算法函式諸如sort和find，使用者經常希望能夠在演算法函式呼叫的附近定義一个临时的述部函式（又称谓词函数，predicate function）。由于語言本身允許在函式內部定義類別，可以考虑使用函数对象，然而這通常既麻煩又冗贅，也阻礙了程式碼的流程。此外，標準C++不允許定義於函式內部的類別被用於模板，所以前述的作法是不可行的。

C++11對lambda的支援可以解決上述問題。

一個lambda函式可以用如下的方式定義：

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

這個不具名函式的回返型別是decltype(x+y)。只有在lambda函式符合"return *expression*"的形式下，它的回返型別才能被忽略。在前述的情況下，lambda函式僅能為一個述句。

在一個更為複雜的例子中，回返型別可以被明確的指定如下：

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

本例中，一個暫時的變數z被建立用來儲存中間結果。如同一般的函式，z的值不會保留到下一次該不具名函式再次被呼叫時。

如果lambda函式沒有傳回值（例如void），其回返型別可被完全忽略。

定義在與lambda函式相同作用域的變數參考也可以被使用。這種的變數集合一般被稱作closure（閉包）。

```
[]       // 沒有定义任何变量。使用未定义变量会引发错误。
[x, &y] // x以传值方式传入（默认），y以引用方式传入。
[&]      // 任何被使用到的外部变量都隐式地以引用方式加以引用。
[=]      // 任何被使用到的外部变量都隐式地以传值方式加以引用。
[&, x]  // x显式地以传值方式加以引用。其余变量以引用方式加以引用。
[=, &z] // z显式地以引用方式加以引用。其余变量以传值方式加以引用。
```

closure被定義與使用如下：

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

上例可計算someList元素的總和並將其印出。變數total是lambda函式closure的一部分，同時它以引用方式被传递入谓词函数，因此它的值可被lambda函式改變。

若不使用引用的符號&，則代表變數以傳值的方式傳入lambda函式。讓使用者可以用這種表示法明確區分變數傳遞的方法：傳值，或是傳參考。由於lambda函式可以不在被宣告的地方就地使用（如置入std::function物件中）；這種情況下，若變數是以傳參考的方式連結到closure中，是無意義甚至是危險的行為。

若lambda函式只在定義的作用域使用，則可以用[&]宣告lambda函式，代表所有引用到stack中的變數，都是以參考的方式傳入，不必一一顯式指明：

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

變數傳入lambda函式的方式可能隨實做有所變化，一般期望的方法是lambda函式能保留其作用域函式的stack指標，藉此存取區域變數。

若使用[=]而非[&]，則代表所有的參考的變數都是傳值使用。

對於不同的變數，傳值或傳參考可以混和使用。比方說，使用者可以讓所有的變數都以傳參考的方式使用，但帶有一個傳值使用的變數：

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

total是傳參考的方式傳入lambda函式，而value則是傳值。

若一個lambda函式被定義於某類別的成員函式中，則可以使用該類別物件的參考，並且能夠存取其內部的成員。

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

這只有當該lambda函式創建的作用域是在SomeType的成員函式內部時才能運作。

在成員函式中指涉物件的this指標，必須要顯式的傳入lambda函式，否則成員函式中的lambda函式無法使用任何該物件的變數或函式。

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

若是lambda函式使用[&]或是[=]的形式，this在lambda函式即為可見。

lambda函式是編譯器從屬型別的函式物件；這種型別名稱只有編譯器自己能夠使用。如果使用者希望將lambda函式作為參數傳入，該型別必須是模版型別，或是必須創建一個`std::function`去獲取lambda的值。使用`auto`關鍵字讓我們能夠儲存lambda函式：

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

返回型別後置的函式宣告

標準C函式宣告語法對於C語言已經足夠。演化自C的C++除了C的基礎語法外，又擴充額外的語法。然而，當C++變得更為複雜時，它暴露出許多語法上的限制，特別是針對函數模板的宣告。下面的範例，不是合法的C++03：

```
template< typename LHS, typename RHS>
  Ret AddingFunc(const LHS &lhs, const RHS &rhs) {return lhs + rhs;} //Ret的型別必須是(lhs+rhs)的型別
```

`Ret`的型別由`LHS`與`RHS`相加之後的結果的型別來決定。即使使用C++11新加入的`decltype`來宣告`AddingFunc`的返回型別，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
```

不合法的原因在於`lhs`及`rhs`在定義前就出現了。直到剖析器解析到函數原型的後半部，`lhs`與`rhs`才是有意義的。

針對此問題，C++11引進一種新的函數定義與聲明的語法：

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

這種語法也能套用到一般的函數定義與聲明：

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

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

關鍵字**auto**的使用與其在自動型別推導代表不同的意義。

物件建構的改良

在標準C++中，建構式不能呼叫其它的建構式；每個建構式必須自己初始化所有的成員或是呼叫一個共用的成員函式。基礎類別的建構式不能夠直接作為衍生類別的建構式；就算基類的建構式已經足夠，每個衍伸的類別仍必須實做自己的建構式。類別中non-constant的資料成員不能夠在宣告的地方被初始化，它們只能在建構式中被初始化。 C++11將會提供這些問題的解決方案。

C++11允許建構式呼叫其他建構式，這種做法稱作委託或轉接（delegation）。僅僅只需要加入少量的代碼，就能讓數個建構式之間達成功能復用（reuse）。Java以及C♯都有提供這種功能。C++11語法如下：

```
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，建構式執行結束代表物件建構完成； 而允許使用轉接建構式的C++11則是以"任何"一個建構式結束代表建構完成。使用委托的建構式，函式本體中的代碼將於被委托的建構式完成後繼續執行（如上例的`PostInit()`）。若基類使用了委托建構式，則衍生類別的建構式會在"所有"基底類別的建構式都完成後，才會開始執行。

C++11允許衍生類別手動繼承基底類別的建構式，編譯器可以使用基底類別的建構式完成衍生類別的建構。而將基類的建構式帶入衍生類的動作，無法選擇性地部分帶入，要不就是繼承基類全部的建構式，要不就是一個都不繼承（不手動帶入）。此外，若牽涉到多重繼承，從多個基底類別繼承而來的建構式不可以有相同的函式簽名（signature）。而衍生類別的新加入的建構式也不可以和繼承而來的基底建構式有相同的函式簽名，因為這相當於重複宣告。

語法如下：

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

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

此語法等同於DerivedClass宣告一個`DerivedClass(int)`的建構式。同時也因為DerivedClass有了一個繼承而來的建構式，所以不會有預設建構式。

另一方面，C++11可以使用以下的語法完成成員初始化：

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

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

若是建構式中沒有設定`iValue`的初始值，則會採用類別定義中的成員初始化，令`iValue`初值為5。在上例中，無參數版本的建構式，`iValue`便採用預設所定義的值；而帶有一個整數參數的建構式則會以指定的值完成初始化。

成員初始化除了上例中的賦值形式（使用"="（，也可以採用建構式以及統一形的初始化（uniform initialization，使用"{}"）。

顯式虛函數重載

在C++裡，在子類別中容易意外的重載虛函數。舉例來說：

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

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

Derived::some_func的真實意圖為何？程序員真的試圖重載該虛函數，或這只是意外？這也可能是base的維護者在其中加入了一個與Derived::some_func同名且擁有相同簽名的虛函式。

另一個可能的狀況是，當基類中的虛函式的簽名被改變，子類中擁有舊簽名的函式就不再重載該虛函式。因此，如果程序員忘記修改所有子類，執行期將不會正確呼叫到該虛函式正確的實現。

C++11將加入支援用來防止上述情形產生，並在編譯期而非執行期捕獲此類錯誤。為保持向後兼容，此功能將是選擇性的。其語法如下：

```
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
};
```

編譯器會檢查基底類別是否存在一虛擬函數，與衍生類別中帶有聲明override的虛擬函數，有相同的函數簽名（signature）；若不存在，則會回報錯誤。

C++11也提供指示字final，用來避免類別被繼承，或是基底類別的函數被改寫：

```
struct Base1 final { };

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

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

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

以上的範例中，virtual void f() final;聲明一新的虛擬函數，同時也表明禁止衍生函數改寫原虛擬函數。

override與final都不是語言關鍵字（keyword），只有在特定的位置才有特別含意，其他地方仍舊可以作為一般指示字（identifier）使用。

空指標

早在1972年，C語言誕生的初期，常數0帶有常數及空指標的雙重身分。 C使用preprocessor macro NULL表示空指標，讓NULL及0分別代表空指標及常數0。 NULL可被定義為((void*)0)或是0。

C++並不採用C的規則，不允許將void*隱式轉換為其他型別的指標。為了使代碼char* c = NULL;能通過編譯，NULL只能定義為0。這樣的決定使得函數多載無法區分代碼的語意：

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

C++建議NULL應當定義為0，所以foo(NULL);將會呼叫foo(int)，這並不是程序員想要的行為，也違反了代碼的直觀性。0的歧義在此處造成困擾。

C++11引入了新的關鍵字來代表空指標常數：nullptr，將空指標和整數0的概念拆開。 nullptr的型別為nullptr_t，能隱式轉換為任何指標或是成員指標的型別，也能和它們進行相等或不等的比較。而nullptr不能隱式轉換為整數，也不能和整數做比較。

為了向下相容，0仍可代表空指標常數。

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

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

強型別列舉

在标准C++中，列舉型別不是型別安全的。列舉型別被視為整數，这使得两种不同的列舉型別之間可以進行比較。C++03唯一提供的安全機制是一个整数或一个枚举型值不能隱式轉換到另一个列舉別型。此外，列舉所使用整數型別及其大小都由實作方法定義，皆無法明確指定。最後，列舉的名稱全數暴露於枚举类型的作用域中，因此兩個不同的列舉，不可以有相同的列舉名。（好比 enum Side{ Right, Left };和 enum Thing{ Wrong, Right };不能一起使用。）

C++11引進了一種特別的"列舉類"，可以避免上述的問題。使用enum class的語法來宣告：

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

此種列舉為型別安全的。列舉類別不能隱式地轉換為整數；也無法與整數數值做比較。（表示式Enumeration::Val4 == 101會觸發編譯期錯誤）。

列舉類別所使用型別必須顯式指定。在上面的範例中，使用的是預設型別int，但也可以指定其他型別：

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

列舉類別的作用域（scoping）不包含枚举值的名字。使用枚举值的名字，必須明確限定于其所屬的枚举类型。例如，前述列舉類別Enum2，Enum2::Val1是有意義的表示法，而單獨的Val1則否。

此外，C++11允許為傳統的列舉指定使用型別：

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

列舉名Val1定義於Enum3的列舉範圍中（Enum3::Val1），但為了向后相容性， Val1仍然可以於所属枚举类型所在的作用域中單獨使用。

在C++11中，列舉類別的前置聲明（forward declaration）也是可行的，只要使用可指定型別的新式列舉即可。之前的C++無法寫出列舉的前置聲明，是由於無法確定列舉變數所佔的空間大小，C++11解決了這個問題：

```
enum Enum1;                     // C++與C++11中不合法;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
```

角括號

標準C++的剖析器一律將">>"視為右移運算子。但在嵌套樣板定義式中，絕大多數的場合其實都代表兩個連續右角括號。為了避免剖析器誤判，撰碼時不能把右角括號連著寫。

C++11變更了剖析器的解讀規則；當遇到連續的右角括號時，会在合理的情况下将右尖括號解析為樣板引數的結束符號。给使用>,>=,>>的表达式加上圆括号，可以避免其与圆括号外部的左尖括号相匹配：

```
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
```

顯式型別轉換子

C++引入了關鍵字explicit来避免用戶自定的單引數建構式被當成隱式型別轉換子。但是，却没有限制明确定义的类型转换函数。比方說，一個smart pointer類別具有一個operator bool()，被定義成若該smart pointer不為null則傳回true，反之傳回false。遇到這樣的代碼時：if(smart_ptr_variable)，編譯器可以藉由operator bool()隱式轉換成布林值，和測试原生指標的方法一樣。但是這類隱式轉換同樣也會發生在非預期之處。由於C++的bool型別也是算术型別，能隱式換為整數甚至是浮點數。拿物件轉換出的布林值做布林運算以外的數學運算，往往不是程序員想要的。

在C++11中，關鍵字explicit修飾符也能套用到型別轉換函数上。如同建構式一樣，它能避免型別轉換函数被隱式轉換調用。但C++11特別指定，在if條件式、迴圈、邏輯運算等需要布林值的地方，將其作为显式类型转换，因此即使对应的类型转换函数被explicit修饰也可以调用。这主要为了解决safe bool问题。

模板的別名

在進入這個主題之前，各位應該先弄清楚「模板」和「型別」本質上的不同。class template (類別模板，是模板)是用來產生template class（模板類別，是型別）。在传统的C++标准，typedef可定義模板類別一個新的型別名稱，但是不能夠使用typedef來定義模板的別名。舉例來說：

```
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
```

這不能夠通過編譯。

為了定義模板的別名，C++11將會增加以下的語法：

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

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

using也能在C++11中定義一般型別的別名，等同typedef：

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

無限制的unions

在C++03中，並非任意的类型都能做為union的成員。比方說，帶有non-trivial 构造函数的型別就不能是union的成員。在新的標準裡，移除了所有对union的使用限制，除了其成員仍然不能是引用型別。這一改變使得union更強大，更有用，也易於使用。[2]

但是如果union成员具有非平凡的特殊成员函数，则编译器不会为union生成对应的特殊成员函数，必须手工定义。

以下為C++11中union使用的簡單样例：

```
struct point
{
  point() {}
  point(int x, int y): x_(x), y_(y) {}
  int x_, y_;
};
union u1
{
     int z;
     double w;
     point p;  // 不合法的C++; point有一non-trivial建構式
               // 合法的C++11

     u1(int x, int y):p(x,y) {};  //Visual Studio 2015 编译通过
};
```

這一改變僅放寬union的使用限制，不會影響既有的舊代碼。

核心語言能力的提升

這些特性让C++語言能夠做一些以前做不到的，或者极其复杂的，或者需求一些不可移植的库的事情。

可变参数模板

在C++11之前，不論是类模板或是函数模板，都只能按其被声明時所指定的样子，接受一組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，不必在定義時將參數的个数固定。

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

模板類tuple的对象，能接受不限個數的typename作為它的模板形参：

```
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。

若不希望產生实参个数為0的不定長參數模板，則可以採用以下的定義：

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

不定長參數模板也能運用到模板函式上。傳統C中的printf函式，雖然也能達成不定長度的引數的調用，但其並非型別安全。以下的範例中，C++11除了能定義型別安全的变长参数函数外，還能讓类似printf的函式能自然地處理非內建型別的物件。除了在模板參數中能使用...表示不定長模板參數外，函數參數也使用同樣的表示法代表不定長參數。

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

其中，Params與parameters分別代表模板與函式的变长参数集合，稱之為參數包（parameter pack）。參數包必須要和运算符"..."搭配使用，避免語法上的歧義。

不定長參數模板中，不定長參數包無法如同一般參數在類或函式中使用； 因此典型的手法是以遞迴的方法取出可用參數，參看以下的C++11 printf範例：

```
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會不斷地遞迴調用自身：函式參數包args...在調用時，會被模板型別匹配分離為T value和Args... args。直到args...變為空參數，則會與簡單的printf(const char *s)形成匹配，結束遞迴。

另一個例子為計算模板參數的長度，這裡使用相似的技巧展開模板參數包Args...：

```
template<typename... args>
struct Count{};
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，但使用算子"..."還能在代碼各處對參數包施以更複雜的展開操作。舉例來說，一個模板類的定義：

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

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

BaseClasses...會被展開成類別ClassName的基底類；ClassName的构造函数需要所有基类的右值引用，而每一個基底類都是以傳入的參數做初始化（BaseClasses(baseClasses)...）。

在函式模板中，不定長參數可以和右值參照搭配，達成引數的完美轉送（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)...));
  }
}
```

參數包parms可展開為TypeToConstruct建構式的引數。表达式std::forward<Args>(params)可將引數的型別信息保留（利用右值參照），傳入建構式。而算子"..."則能將前述的表示式套用到每一個參數包中的參數。這種工廠函式（factory function）的手法，使用std::shared_ptr管理配置物件的記憶體，避免了不當使用所產生的記憶體洩漏（memory leaks）。

此外，不定長參數的數量可以藉以下的語法得知：

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

SomeStruct<Type1, Type2>::size是2，而SomeStruct<>::size會是0。（sizeof...(Args)的結果是編譯期常數。）

新的字串字面值

標準C++提供了兩種字串字面值。第一種，包含有雙引號，產生以空字元結尾的const char陣列。第二種有著前標L，產生以空字元結尾的const wchar_t陣列，其中wchar_t代表寬字元。對於Unicode編碼的支援尚付闕如。

為了加強C++編譯器對Unicode的支援，型別char的定義被修改為其大小至少能夠儲存UTF-8的8位元編碼，並且能夠容納編譯器的基本字元集的任何成員。

C++11將支援三種Unicode編碼方式：UTF-8，UTF-16，和UTF-32。除了上述char定義的變更，C++11將增加兩種新的字元型別：char16_t和char32_t。它們各自被設計用來儲存UTF-16以及UTF-32的字元。

以下展示如何產生使用這些編碼的字串字面值：

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

第一個字串的型別是通常的const char[]；第二個字串的型別是const char16_t[]；第三個字串的型別是const char32_t[]。

當建立Unicode字串字面值時，可以直接在字串內插入Unicode codepoints。C++11提供了以下的語法：

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

在'\u'之後的是16個位元的十六進位數值；它不需要'0x'的前標。識別字'\u'代表了一個16位元的Unicode codepoint；如果要輸入32位元的codepoint，使用'\U'和32個位元的十六進位數值。只有有效的Unicode codepoints能夠被輸入。舉例而言，codepoints在範圍U+D800—U+DFFF之間是被禁止的，它們被保留給UTF-16編碼的surrogate pairs。

有時候避免手動將字串換碼也是很有用的，特別是在使用XML檔案或是一些腳本語言的字面值的時候。C++11將提供raw（原始）字串字面值：

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

在第一個例子中，任何包含在( )括號（標準已經從[]改為()）當中的都是字串的一部分。其中"和\字元不需要經過转义。在第二個例子中，"delimiter(開始字串，只有在遇到)delimiter"才代表結束。其中delimiter可以是最多16个字符的任意的字串（包含空字符串），但不能包含空格、控制字符和'('、')'、'\'。原始字符串

允許使用者使用圆括号(,)，例如R"delimiter((a-z))delimiter"等价于"(a-z)"。原始字串字面值能夠和寬字面值或是Unicode字面值結合：

```
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.)"
```

用户定义字面量

標準C++提供了數種字面值。字元"12.5"是能夠被編譯器解釋為數值12.5的`double`型別字面值。然而，加上"f"的後置，像是"12.5f"，則會產生數值為12.5的`float`型別字面值。之前的C++規範中字面值的修饰符是固定的，C++代碼不能創立新的字面修饰符。

C++11開放使用者定義新的字面修飾符（literal modifier），利用自訂的修飾符完成由字面值建構物件。

字面值轉換可以定义為兩個階段：原始與轉換後（raw與cooked）。原始字面值指特定类型的字符序列，而轉換後的字面值則代表另一種型別。如字面值`1234`，原始字面值是`'1', '2', '3', '4'`的字符序列；而轉換後的字面值是整數值`1234`。另外，字面值`0xA`轉換前是序列`'0', 'x', 'A'`；轉換後代表整數值10。

多工記憶體模型

C++标准委员会计划统一对多緒編程的支援。

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、为线程之間的互動提供支援。第二部分将由程式庫提供支持，更多请看緒程支援。

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最佳化和記憶體一致性的问题。

thread-local 的存储期限

在多緒環境下，讓各緒程擁有各自的變數是很普遍的。這已經存在於函式的區域變數，但是對於全域和靜態變數都還不行。

新的*thread_local*存儲期限（在現行的*static*、*dynamic*和*automatic*之外）被作為下個標準而提出。緒程區域的存儲期限會藉由存儲指定字`thread_local`來表明。

static物件(生命週期為整個程式的執行期間)的存儲期限可以被thread-local給替代。就如同其他使用static存儲期的變數，thread-local物件能夠以建構式初始化並以解構式摧毀。

使用或禁用物件的預設函式

在傳統C++中，若使用者沒有提供，則編譯器會自動為物件生成預設建構式(default constructor)、複製建構式（copy constructor），賦值運算子（copy assignment operator `operator=`）以及解構式（destructor）。另外，C++也為所有的類別定義了數個全域算子（如`operator delete`及`operator new`）。當使用者有需要時，也可以提供自訂的版本改寫上述的函式。

問題在於原先的c++無法精確地控制這些預設函數的生成。比方說，要讓類別不能被拷貝，必須將複製建構式與賦值運算子宣告為private，並不去定義它們。嘗試使用這些未定義的函式會導致編譯期或連結期的錯誤。但這種手法並不是一個理想的解決方案。

此外，編譯器產生的預設建構式與使用者定義的建構式無法同時存在。若使用者定義了任何建構式，編譯器便不會生成預設建構式； 但有時同時帶有上述兩者提供的建構式也是很有用的。目前並沒有顯式指定編譯器產生預設建構式的方法。

C++11允許顯式地表明採用或拒用編譯器提供的內建函式。例如要求類別帶有預設建構式，可以用以下的語法：

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

另一方面，也可以禁止編譯器自動產生某些函式。如下面的例子，類別不可複製：

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

禁止類別以operator new配置記憶體：

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

此種物件只能生成於stack中或是當作其他類別的成員，它無法直接配置於heap之中，除非使用了與平台相關，不可移植的手法。（使用placement new算子雖然可以在用戶自配置的記憶體上呼叫物件建構式，但在此例中其他形式的new算子一併被上述的定義遮蔽（"name hiding"），所以也不可行。）

= delete的聲明（同時也是定義）也能適用於非內建函式，禁止成員函式以特定的引數呼叫：

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

若嘗試以double的引數呼叫f()，將會引發編譯期錯誤，編譯器不會自動將double引數轉型為int再呼叫f()。若要徹底的禁止以非int的引數呼叫f()，可以將= delete與模板相結合：

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

long long int型別

在32位元系統上，一個long long int是保有至少64個有效位元的整數型別。C99將這個型別引入了標準C中，目前大多數的C++編譯器也支援這種型別。C++11將把這種型別添加到標準C++中。

靜態assertion

C++提供了兩種方法測試assertion（聲明）：巨集assert以及前處理器指令#error。但是這兩者對於模版來說都不合用。巨集在執行期測試assertion，而前處理器指令則在前置處理時測試assertion，這時候模版還未能具現化。所以它們都不適合來測試牽扯到模板參數的相關特性。

新的機能會引進新的方式可以在編譯期測試assertion，只要使用新的關鍵字static_assert。宣告採取以下的形式：

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

這裡有一些如何使用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!"  ) ;
} ;
```

當常數表示式值為false時，編譯器會產生相應的錯誤訊息。第一個例子是前處理器指令#error的替代方案；第二個例子會在每個模板類別Check生成時檢查assertion。

靜態assertion在模板之外也是相當有用的。例如，某個演算法的實作依賴於long long型別的大小比int還大，這是標準所不保證的。這種假設在大多數的系統以及編譯器上是有效的，但不是全部。

允許sizeof運算子作用在類別的資料成員上，無須明確的物件

在標準C++，sizeof可以作用在物件以及型別上。但是不能夠做以下的事：

```
struct SomeType { OtherType member; };

sizeof(SomeType::member); // 直接由SomeType型別取得非靜態成員的大小，C++03不行。C++11允許
```

這會傳回OtherType的大小。C++03並不允許這樣做，所以會引發編譯錯誤。C++11將會允許這種使用。

垃圾回收機制

是否會自動回收那些無法被使用到（unreachable）的動態分配物件由實作決定。

C++標準程式庫的變更

C++11標準程式庫有數個新機能。其中許多可以在現行標準下實作，而另外一些則依賴於（或多或少）新的C++11核心語言機能。

新的程式庫的大部分被定義於*C++標準委員會的Library Technical Report*（稱TR1），於2005年發布。各式TR1的完全或部分實作目前提供在命名空間std::tr1。C++11會將其移置於命名空間std之下。

標準庫元件上的升級

目前的標準庫能受益於C++11新增的一些語言特性。舉例來說，對於大部份的標準庫容器而言，像是搬移內含大量元素的容器，或是容器之內對元素的搬移，基於右值引用（Rvalue reference）的move建構子都能優化前述動作。在適當的情況下，標準庫元件將可利用C++11的語言特性進行升級。這些語言特性包含但不局限以下所列：

- 右值引用和其相關的move支援
- 支援UTF-16編碼，和UTF-32字元集
- 變長參數模板（與右值引用搭配可以達成完美转发（perfect forwarding））
- 編譯期常數表達式
- Decltype
- 顯式型別轉換子
- 使用或禁用物件的預設函式

此外，自C++標準化之後已經過許多年。現有許多代碼利用到了標準庫;這同時揭露了部份的標準庫可以做些改良。其中之一是標準庫的記憶體配置器（allocator）。C++11將會加入一個基於作用域模型的記憶體配置器來支援現有的模型。

執行緒支援

雖然C++11會在語言的定義上提供一個記憶體模型以支援執行緒，但執行緒的使用主要將以C++11標準函式庫的方式呈現。

C++11標準函式庫會提供類別`thread`（`std::thread`）。若要執行一個執行緒，可以建立一個類別`thread`的實體，其初始參數為一個函式物件，以及該函式物件所需要的參數。透過成員函式`std::thread::join()`對執行緒會合的支援，一個執行緒可以暫停直到其它執行緒執行完畢。若有底層平台支援，成員函式`std::thread::native_handle()`將可提供對原生執行緒物件執行平台特定的操作。

對於執行緒間的同步，標準函式庫將會提供適當的互斥鎖（像是`std::mutex`，`std::recursive_mutex`等等）和條件變數（`std::condition_variable`和`std::condition_variable_any`）。前述同步機制將會以RAII鎖（`std::lock_guard`和`std::unique_lock`）和鎖相關演算法的方式呈現，以方便程式員使用。

對於要求高效能，或是極底層的工作，有時或甚至是必須的，我們希望執行緒間的通訊能避免互斥鎖使用上的開銷。以原子操作來存取記憶體可以達成此目的。針對不同情況，我們可以透過顯性的記憶體屏障改變該存取記憶體動作的可見性。

對於執行緒間非同步的傳輸，C++11標準函式庫加入了以及`std::packaged_task`用來包裝一個會傳回非同步結果的函式呼叫。因為缺少結合數個future的功能，和無法判定一組promise集合中的某一個promise是否完成，futures此一提案因此而受到了批評。

更高級的執行緒支援，如執行緒池，已經決定留待在未來的Technical Report加入此類支援。更高級的執行緒支援不會是C++11的一部份，但設想是其最終實現將建立在目前已有的執行緒支援之上。

`std::async`提供了一個簡便方法以用來執行執行緒，並將執行緒綁定在`std::future`。使用者可以選擇一個工作是要多個執行緒上非同步的執行，或是在一個執行緒上執行並等待其所需要的資料。預設的情況，實作可以根據底層硬體選擇前面兩個選項的其中之一。另外在較簡單的使用情形下，實作也可以利用執行緒池提供支援。

多元組型別

多元組是一個內由數個異質物件以特定順序排列而成的資料結構。多元組可被視為是`struct`其資料成員的一般化。

由TR1演進而來的C++11多元組型別將受益於C++11某些特色像是可變參數模板。TR1版本的多元組型別對所能容納的物件個數會因實作而有所限制，且實作上需要用到大量的巨集技巧。相反的，C++11版本的多元組型基本上於對其能容納的物件個數沒有限制。然而，編譯器對於模板實體化的遞迴深度上的限制仍舊影響了元組型別所能容納的物件個數（這是無法避免的情況）；C++11版本的多元組型不會把這個值讓使用者知道。

使用可變參數模板，多元組型別的宣告可以長得像下面這樣：

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

底下是一個多元組型別的定義和使用情況：

```
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);  // 將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
```

我們可以定義一個多元組型別物件proof而不指定其內容，前提是proof裡的元素其型別定義了預設建構子（default constructor）。此外，以一個多元組型別物件賦值給另一個多元組型別物件是可能的，但只有在以下情況：若這兩個多元組型別相同，則其內含的每一個元素其型別都要定義拷貝建構子（copy constructor）；否則的話，賦值操作符右邊的多元組其內含元素的型別必須能轉換成左邊的多元組其對應的元素型別，又或者賦值操作符左邊的多元組其內含元素的型別必須定義適當的建構子。

```
std::tuple< int , double, string       > t1;
std::tuple< char, short , const char * > 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。
```

多元組類型物件的比較運算是可行的（當它們擁有同樣數量的元素）。此外，C++11提供兩個表達式用來檢查多元組類型的一些特性（僅在編譯期做此檢查）。

- std::tuple_size<T>::value回傳多元組T內的元素個數，
- std::tuple_element<I, T>::type回傳多元組T內的第I個元素的型別

雜湊表

在過去，不斷有要求想將雜湊表（無序關聯式容器）引進標準庫。只因為時間上的限制，雜湊表才沒有被標準庫所採納。雖然，雜湊表在最糟情況下（如果出現許多衝突（collision）的話）在效能上比不過平衡樹。但實際運用上，雜湊表的表現則較佳。

因為標準委員會還看不到有任何機會能將開放定址法標準化，所以目前衝突僅能透過鏈地址法（linear chaining）的方式處理。為避免與第三方函式庫發展的雜湊表發生名稱上的衝突，字首將採用unordered而非hash。

函式庫將引進四種雜湊表，其中差別在於底下兩個特性：是否接受具相同鍵值的項目（Equivalent keys），以及是否會將鍵值映射到相對應的資料（Associated values）。

雜湊表類型	有無關聯值	接受相同鍵值
std::unordered_set	否	否
std::unordered_multiset	否	是
std::unordered_map	是	否
std::unordered_multimap	是	是

上述的類別將滿足對一個容器類別的要求，同時也提供存取其中元素的成員函式：insert，erase，begin，end。

雜湊表不需要對現有核心語言做擴展（雖然雜湊表的實作會利用到C++11新的語言特性），只會對標頭檔<functional>做些許擴展，並引入<unordered_set>和<unordered_map>兩個標頭檔。對於其它現有的類別不會有任何修改。同時，雜湊表也不會依賴其它標準庫的擴展功能。

正規表示式

過去許多或多或少標準化的程式庫被建立用來處理正規表示式。有鑑於這些演算法的使用非常普遍，因此標準程式庫將會包含他們，並使用各種物件導向語言的潛力。

這個新的程式庫，被定義於<regex>標頭檔，由幾個新的類別所組成：

- 正規表示式（樣式）以樣板類basic_regex的實體表示
- 樣式匹配的情況以樣板類match_results的實體表示

函式regex_search是用來搜尋樣式；若要搜尋並取代，則要使用函式regex_replace，該函式會回傳一個新的字串。演算法regex_search和regex_replace接受一個正規表示式（樣式）和一個字串，並將該樣式匹配的情況儲存在struct match_results。

底下描述了match_results的使用情況：

```
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" ;
  }
}
```

注意雙反斜線的使用，因為C++將反斜線作為跳脫字元使用。但C++11的raw string可以用來避免此一問題。函式庫<regex>不需要改動到現有的標頭檔，同時也不需要對現有的語言作擴展。

通用智能指針

這些指針是由TR1智能指針演變而來。注意! 智能指針是類別而非一般指針。

shared_ptr是一引用計數（reference-counted）指針，其行為與一般C++指針極為相似。在TR1的實作中，缺少了一些一般指針所擁有的特色，像是別名或是指針運算。C++11新增前述特色。

一個shared_ptr只有在已經沒有任何其它shared_ptr指向其原本所指向物件時，才會銷毀該物件。

一個weak_ptr指向的是一個被shared_ptr所指向的物件。該weak_ptr可以用來決定該物件是否已被銷毀。weak_ptr不能被解參考;想要存取其內部所保存的指針，只能透過shared_ptr。有兩種方法可達成此目的。第一，類別shared_ptr有一個以weak_ptr為參數的建構子。第二，類別weak_ptr有一個名為lock的成員函式，其返回值為一個shared_ptr。weak_ptr並不擁有它所指向的物件，因此不影響該物件的銷毀與否。

底下是一個shared_ptr的使用範例：

```
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將會被C++標準所廢棄，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、隱性的左值搬移。不像auto_ptr，unique_ptr可以存放在C++11提出的那些能察覺搬移動作的容器之中。

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。不過其演算法則取決於各程式庫開发者。C++直接從C繼承了這部份，但是C++11將會提供產生偽亂數的新方法。

C++11的随机数功能分为两部分：第一，一個亂數生成引擎，其中包含該生成引擎的狀態，用來產生亂數。第二，一個分布，這可以用來決定產生亂數的範圍，也可以決定以何種分布方式產生亂數。亂數生成物件即是由亂數生成引擎和分布所構成。

不同於C標準庫的rand;針對產生亂數的機制，C++11將會提供三種演算法，每一種演算法都有其強項和弱項：

樣板類	整數/浮點數	品質	速度	狀態數*
linear_congruential	整數	低	中等	1
subtract_with_carry	兩者皆可	中等	快	25
mersenne_twister	整數	佳	快	624

C++11將會提供一些標準分布：uniform_int_distribution（離散型均勻分佈），bernoulli_distribution(伯努利分布），geometric_distribution（幾何分佈），poisson_distribution（卜瓦松分佈），binomial_distribution（二項分佈），uniform_real_distribution（離散型均勻分佈)，exponential_distribution（指数分布），normal_distribution（常態分佈）和gamma_distribution（伽玛分布）。

底下描述一個亂數生成物件如何由亂數生成引擎和分布構成：

```
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99之間
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

我們可以透過實體化樣板類reference_wrapper得到一個包装引用（wrapper reference）。包装引用類似於一般的引用。對於任意物件，我們可以透過模板類ref得到一個包装引用（至於constant reference則可透過cref得到）。

當樣板函式需要形參的引用而非其拷貝，這時包装引用就能派上用場：

```
// 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_wrapp er<int>>'
                         // 'i'會被修改
    std::cout << i << std::endl;  // 輸出1
}
```

這項功能將加入標頭檔<utility>之中，而非透過擴展語言來得到這項功能。

多态函数对象包装器

針對函数对象的多态包装器（又稱多态函数对象包装器）在語義和語法上和函式指標相似，但不像函式指標那麼狹隘。只要能被呼叫，且其參數能與包裝器相容的都能以多态函数对象包装器稱之（函式指標，成員函式指標或仿函式）。

透過以下例子，我們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
```

模板類function將定義在標頭檔<functional>，而不須更動到語言本身。

用於元編程的型別屬性

對於那些能自行創建或修改本身或其它程式的程式，我們稱之為元編程。這種行為可以發生在編譯或執行期。C++標準委員會已經決定引進一組由模板實現的函式庫，程式員可利用此一函式庫於編譯期進行元編程。

底下是一個以元編程來計算指數的例子：

```
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;
```

許多演算法能作用在不同的資料型別；C++模板支援泛型，這使得代碼能更緊湊和有用。然而，演算法經常會需要目前作用的資料型別的資訊。這種資訊可以透過型別屬性（type traits）於模板實體化時將該資訊萃取出來。

型別屬性能識別一個物件的種類和有關一個型別（class或struct）的特徵。標頭檔<type_traits>描述了我們能識別那些特徵。

底下的例子說明了模板函式‘elaborate’是如何根據給定的資料型別，從而實體化某一特定的演算法（algorithm.do_it）。

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

// 演算法二
template<> struct Algorithm<true> {
    template<class T1, class T2> static 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 ) ;
}
```

此種編程技巧能寫出優美、簡潔的代碼；然而除錯是此種編程技巧的弱處：編譯期的錯誤訊息讓人不知所云，執行期的除錯更是困難。

用于计算函数对象返回类型的统一方法

要在編譯期決定一個樣板仿函式的回返值型別並不容易，特別是當回返值依賴於函式的參數時。舉例來說：

```
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;
};
```

實體化樣板類`Calculus<Clear>`，`Calculus`的仿函式其回返值總是和`Clear`的仿函式其回返值具有相同的型別。然而，若給定類別`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
};
```

企圖實體化樣板類Calculus<Confused>將導致Calculus的仿函式其回返值和類別Confused的仿函式其回返值有不同的型別。對於`int`和`double`之間的轉換，編譯器將給出警告。

模板`std::result_of`被TR1引進且被C++11所採納，可允許我們決定和使用一個仿函式其回返值的型別。底下，`CalculusVer2`物件使用`std::result_of`物件來推導其仿函式的回返值型別：

```
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;
};
```

如此一來，在實體化`CalculusVer2<Confused>`其仿函式時，不會有型別轉換，警告或是錯誤發生。

模板`std::result_of`在TR1和C++11有一點不同。TR1的版本允許實作在特殊情況下，可以無法決定一個函式呼叫其回返值型別。然而，因為C++11支持了decltype，實作被要求在所有情況下，皆能計算出回返值型別。

iota 函数

iota 函数可将给定区间的值设定为从某值开始的连续值，例如将连续十个整数设定为从 1 开始的连续整数（即 1、2、3、4、5、6、7、8、9、10）。

```
#include <iostream>
#include <array>
#include <numeric>

std::array<int, 10> ai;
std::iota(ai.begin(), ai.end(), 1);
for(int i: ai){
  std::cout<<i<<" ";//1 2 3 4 5 6 7 8 9 10
}
```

已被移除或是不包含在 C++11標準的特色

預計由Technical Report提供支援：

- 模組
- 十進制型別
- 数学专用函数

延後討論：

- Concepts（概念 (C++)）
- 更完整或必備的垃圾回收支援
- Reflection
- Macro Scopes

被移除或廢棄的特色

- 循序點（sequence point），這個術語正被更為易懂的描述所取代。一個運算可以發生（is sequenced before）在另一個運算之前;又或者兩個運算彼此之間沒有順序關係（are unsequenced）。
- export
- exception specifications
- std::auto_ptr被**std::unique_ptr**取代。
- 仿函式基底類別 (std::unary_function, std::binary_function)、函式指針適配器、類別成員指針適配器以及綁定器 (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
- Visual C++ 2010与Visual C++ 2012支持的C++11特性的对比列表：C++11 Features (Modern C++)
- Visual C++ 2013支持的C++11特性的对比列表：C++11 Features (Modern C++)
- Visual C++ 2015预览版支持的C++11/14/17特性的对比列表：VC2015 Preview语言特性列表
- 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

關聯項目

- C++ Technical Report 1
- C11，C程式語言的最新標準
- C++14，C++的最新標準

參考資料

1. N3376 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf)
2. N2544 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2544.pdf)

C++標準委員會文件

- ^ Doc No. 1401: Jan Kristoffersen（2002年10月21日）*Atomic operations with multi-threaded environments*
- ^ Doc No. 1402: Doug Gregor（2002年10月22日）*A Proposal to add a Polymorphic Function Object Wrapper to the Standard Library*
- ^ Doc No. 1403: Doug Gregor（2002年11月8日）*Proposal for adding tuple types into the standard library*
- ^ Doc No. 1424: John Maddock（2003年3月3日）*A Proposal to add Type Traits to the Standard Library*
- ^ Doc No. 1429: John Maddock（2003年3月3日）*A Proposal to add Regular Expression to the Standard Library*
- ^ Doc No. 1449: 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: 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: Jens Maurer (April 10, 2003) *A Proposal to Add an Extensible Random Number Facility to the Standard Library (Revision 2)*
- ^ Doc No. 1453: D. Gregor, P. Dimov (April 9, 2003) *A proposal to add a reference wrapper to the standard library (revision 1)*
- ^ Doc No. 1454: Douglas Gregor, P. Dimov (April 9, 2003) *A uniform method for computing function object return types (revision 1)*
- ^ Doc No. 1456: Matthew Austern (April 9, 2003) *A Proposal to Add Hash Tables to the Standard Library (revision 4)*
- ^ Doc No. 1471: Daveed Vandevoorde (April 18, 2003) *Reflective Metaprogramming in C++*
- ^ Doc No. 1676: Bronek Kozicki (September 9, 2004) *Non-member overloaded copy assignment operator*
- ^ Doc No. 1704: Douglas Gregor, Jaakko Järvi, Gary Powell (September 10, 2004) *Variadic Templates: Exploring the Design Space*
- ^ Doc No. 1705: J. Järvi, B. Stroustrup, D. Gregor, J. Siek, G. Dos Reis (September 12, 2004) *Decltype (and auto)*
- ^ Doc No. 1717: Francis Glassborow, Lois Goldthwaite (November 5, 2004) *explicit class and default definitions*
- ^ Doc No. 1719: Herb Sutter, David E. Miller (October 21, 2004) *Strongly Typed Enums (revision 1)*
- ^ Doc No. 1720: 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: Daveed Vandevoorde (January 14, 2005) *Right Angle Brackets (Revision 2)*
- ^ Doc No. 1811: J. Stephen Adamczyk (April 29, 2005) *Adding the long long type to C++ (Revision 3)*
- ^ Doc No. 1815: Lawrence Crowl (May 2, 2005) *ISO C++ Strategic Plan for Multithreading*
- ^ Doc No. 1827: Chris Uzdavinis, Alisdair Meredith (August 29, 2005) *An Explicit Override Syntax for C++*
- ^ Doc No. 1834: Detlef Vollmann (June 24, 2005) *A Pleading for Reasonable Parallel Processing Support in C++*
- ^ Doc No. 1836: ISO/IEC DTR 19768 (June 24, 2005) *Draft Technical Report on C++ Library Extensions*
- ^ Doc No. 1886: Gabriel Dos Reis, Bjarne Stroustrup (October 20, 2005) *Specifying C++ concepts*
- ^ Doc No. 1891: Walter E. Brown (October 18, 2005) *Progress toward Opaque Typedefs for C++0X*
- ^ Doc No. 1898: Michel Michaud, Michael Wong (October 6, 2004) *Forwarding and inherited constructors*
- ^ Doc No. 1919: Bjarne Stroustrup, Gabriel Dos Reis (December 11, 2005) *Initializer lists*
- ^ Doc No. 1968: V Samko; J Willcock, J Järvi, D Gregor, A Lumsdaine (February 26, 2006) *Lambda expressions and closures for C++*
- ^ Doc No. 1986: Herb Sutter, Francis Glassborow (April 6, 2006) *Delegating Constructors (revision 3)*
- ^ Doc No. 2016: Hans Boehm, Nick Maclaren (April 21, 2002) *Should volatile Acquire Atomicity and Thread Visibility Semantics?*

- ^ Doc No. 2142: ISO/IEC DTR 19768 (January 12, 2007) *State of C++ Evolution (between Portland and Oxford 2007 Meetings)*
- ^ Doc No. 2228: ISO/IEC DTR 19768 (May 3, 2007) *State of C++ Evolution (Oxford 2007 Meetings)*
- ^ Doc No. 2258: G. Dos Reis and B. Stroustrup *Templates Aliases*
- ^ Doc No. 2280: Lawrence Crowl (May 2, 2007) *Thread-Local Storage*
- ^ Doc No. 2291: ISO/IEC DTR 19768 (June 25, 2007) *State of C++ Evolution (Toronto 2007 Meetings)*
- ^ Doc No. 2336: ISO/IEC DTR 19768 (July 29, 2007) *State of C++ Evolution (Toronto 2007 Meetings)*
- ^ Doc No. 2389: ISO/IEC DTR 19768 (August 7, 2007) *State of C++ Evolution (pre-Kona 2007 Meetings)*
- ^ Doc No. 2431: SC22/WG21/N2431 = J16/07-0301 (October 2, 2007), *A name for the null pointer: nullptr*
- ^ Doc No. 2432: ISO/IEC DTR 19768 (October 23, 2007) *State of C++ Evolution (post-Kona 2007 Meeting)*
- ^ Doc No. 2437: Lois Goldthwaite (October 5, 2007) *Explicit Conversion Operators*
- ^ Doc No. 2461: ISO/IEC DTR 19768 (October 22, 2007) *Working Draft, Standard for programming Language C++*
- ^ Doc No. 2507: ISO/IEC DTR 19768 (February 4, 2008) *State of C++ Evolution (pre-Bellevue 2008 Meeting)*
- ^ Doc No. 2544: Alan Talbot, Lois Goldthwaite, Lawrence Crowl, Jens Maurer (February 29, 2008) *Unrestricted unions*
- ^ Doc No. 2565: ISO/IEC DTR 19768 (March 7, 2008) *State of C++ Evolution (post-Bellevue 2008 Meeting)*
- ^ Doc No. 2597: ISO/IEC DTR 19768 (April 29, 2008) *State of C++ Evolution (pre-Antipolis 2008 Meeting)*
- ^ Doc No. 2606: ISO/IEC DTR 19768 (May 19, 2008) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 2697: ISO/IEC DTR 19768 (June 15, 2008) *Minutes of WG21 Meeting June 8–15, 2008*
- ^ Doc No. 2798: ISO/IEC DTR 19768 (October 4, 2008) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 2857: ISO/IEC DTR 19768 (March 23, 2009) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 2869: ISO/IEC DTR 19768 (April 28, 2009) *State of C++ Evolution (post-San Francisco 2008 Meeting)*
- ^ Doc No. 3000: ISO/ISC DTR 19769 (November 9, 2009) *Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3014: 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: Herb Sutter（2010年3月13日）*C++0x Meeting Schedule*
- ^ Doc No. 3092: ISO/ISC DTR 19769（2010年3月26日）*Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3126: ISO/ISC DTR 19769（2010年8月21日）*Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3225: ISO/ISC DTR 19769（2010年11月27日）*Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3242: ISO/ISC DTR 19769（2011年2月28日）*Working Draft, Standard for Programming Language C++*
- ^ Doc No. 3290: 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
- DevX.com Special Report（2008年8月18日）C++0x: The Dawning of a New Standard

外部連結

- **The C++ Standards** Committee
- Bjarne Stroustrup's homepage
- C++0X: The New Face of Standard C++
- Herb Sutter's blog coverage of C++0X
- A talk on C++0x given by Bjarne Stroustrup at the University of Waterloo
- A quick and dirty introduction to C++0x (as of November 2007)
- The State of the Language: An Interview with Bjarne Stroustrup (August 15, 2008)
- Working draft for the C++ language, October 4, 2008