# C++

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 版本 | 修订内容 | 修改人 |
| 2018-09-30 | V0.1 | 初稿 | 朱灿基 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 一、概述

“凡事豫则立，不豫则废”

C++11相比C++0x有较大的改动，添加了很多新特性。

# 二、C++基础

## 2.1、内存

### 2.2.1、new/delete

free/delete怎么知道有多少内存要释放？[[2]、[3]](#_参考文献)

实际上在申请内存的时候，申请到的地址会比你实际的地址大一点点，它包含了一个存有申请空间大小的结构体。

比如你申请了20byte的空间，实际上系统申请了48byte的空间。

|  |
| --- |
| 16-byte header containing size, special marker, checksum, pointers to next/previous block and so on.  32 bytes data area (your 20 bytes padded out to a multiple of 16)) |

The address then given to you is the address of the data area. Then, when you free the block, free will simply take the address you give it and, assuming you haven't stuffed up that address or the memory around it, check the accounting information immediately before it. Graphically, that would be along the lines of:

|  |
| --- |
| \_\_\_\_ The allocated block \_\_\_\_  / \  +--------+--------------------+  | Header | Your data area ... |  +--------+--------------------+  ^  |  +-- The address you are given |

Keep in mind the size of the header and the padding are totally implementation defined (actually, the entire thing is implementation-defined (a) but the in-line accounting option is a common one).

The checksums and special markers that exist in the accounting information are often the cause of errors like "Memory arena corrupted" or "Double free" if you overwrite them or free them twice.

The padding (to make allocation more efficient) is why you can sometimes write a little bit beyond the end of your requested space without causing problems (still, don't do that, it's undefined behaviour and, just because it works sometimes, doesn't mean it's okay to do it).

这样就可以不需要提供任何其他信息，而正确的释放内存。

## 2.2、指针

### 2.2.1、指针变量的定义

在定义指针变量时必须指定基类型。为什么要指定基类型呢？

因为不同类型的数据在内存中占用的字节和存放方式是不一样的（例如整型占8个字节，字符型占1个字节）。如果想通过指针引用一个变量，只知道一个地址是不够的，因为无法判断从地址处取多少个字节的数据。必须知道数据的类型，才能按照存储单元的长度以及数据的存储形式正确的取出该数据。

指针的加减运算是改变多少个数据类型的长度。

比如，“使指针值加1”，对于整型指针（int \*）这个1意味着使地址值加8个字节。（假设是64位机器）

通过指针引用数组。当指针指向数组元素的时候，指针变量p指向数组元素a[0]，用p+1表示指向下一个元素a[1]。

### 2.2.2、指向函数的指针

函数指针：如果在程序中定义了一个函数，在编译时，编译系统位函数代码分配了一段存储空间，这段存储空间的起始地址（又称为入口地址）称为这个函数的指针。

可以定义一个指向函数的指针变量，用来存放某一函数的起始地址，这就意味着此指针变量指向该函数。例如

|  |
| --- |
| int (\* p)(int , int); |

定义p是一个指向函数的指针变量，它可以指向函数的类型位整型且有两个整型参数的函数。p的类型用int (\*)(int , int)表示。

用函数指针变量调用函数

|  |
| --- |
| int max(int, int);  int (\*p)(int, int);//定义指向函数的指针变量  **p = max;**//使指针变量指向该函数  int a, b, c;  **c = (\*p)(a, b);**//用函数指针变量调用函数，用(\*p)代替函数名。 |

### 2.2.3、指针数组

一个数组，若其元素均为指针类型数据，称之为指针数组。也就是说，指针数组中的每一个元素都存放一个地址，相当于一个指针变量。

|  |
| --- |
| int \* p[4]; |

由于[]比\*优先级高，因此p先与[4]结合，形参p[4]形式，这显然是数组形式，表示p数组有4个元素。然后再与p前面的”\*”结合，“\*”表示此数组是指针类型的，每个数组元素都可指向一个整形变量。

指针数组比较适合用来指向若干个字符串，使字符串处理更加方便灵活。

指向指针数据的指针：指向指针数据的指针变量，简称为指向指针的指针。

## 2.3、值传递和引用传递

当类类型作为函数参数进行值传递和引用传递时，构造函数、析构函数、拷贝构造函数调用次数及资源问题。

### 2.3.1、函数参数中值传递和引用传递

### 2.3.2、函数返回值的值传递和引用传递

### 2.3.3、小结

## 2.4、深复制和浅复制

# 三、深入理解C++11：C++11新特性解析与应用

C++11相对于C++98/03有那些显著的增强呢？

1、通过内存模型、线程、原子操作等来支持本地并行编程（Native Concurrency）。

2、通过统一初始化表达式、auto、declytype、移动语义等来统一对泛型编程的支持。

3、通过constexpr、POD（概念）等更好地支持系统编程。

4、通过内联命名空间、继承构造函数和右值引用等，以更好地支持库的构建。

C++11就像是一门新的语言。如果将C++98标准中的特性和C++11放到一起，C++11就像是个恐怖的“编程语言泛型联盟”。利用它不仅仅可以写出面向对象语言的代码，也可以写出过程式编程语言代码、泛型编程语言代码、函数式编程语言代码、元编程编程语言代码，或者其它。多泛型的支持使得C++11语言的“硬能力”几乎在编程语言中”无出其右“。

C++标准委员会更专注于以下理念：

1、保持语言的稳定性和兼容性（Maintain stability and compatibility）

2、更倾向于使用库而不是扩展语言来实现特性（Prefer libraries to language extensions）。

3、更倾向于通用的而不是特殊的手段来实现特性（Prefer generality to specialization）。

4、专家新手一概支持（Support both experts and novices）。

5、增强类型的安全性（Increase type safety）。

6、增强代码执行性能和操作硬件的能力（Improve performance and ability to work directly）。

7、开发能够改变人们思维方式的特性（Make only changes that change the way people think）。

8、融入编程现实（Fit into the real world）.

设计理念中“更倾向于使用库而不是扩展语言来实现特性“的部分。库有一个很大的优势，就是其改动不需要编译器实现新特性（只要接口保持一致即可），当然，更重要的是库可以用于支持不同领域的编程。这样一来，通常读者不需要非常精通C++就能使用它们。

C++语言可以尽量保持较少的核心语言特性的同时，通过标准库扩大其功能。

下面列出了属于该设计理念下的库特性：

1、算法增强Algorithm improvements

2、容器增强Container improvements

3、分配算符Scoped allocators

4、std::array

5、std::forward\_list

6、无序容器Unordered containers

7、std::tuple

8、类型特性Type traits

9、std::function, std::bind

10、unique\_ptr

11、shared\_ptr

12、weak\_ptr

13、线程Threads

14、互斥Mutex

15、锁Locks

16、条件变量Condition variables

17、时间工具Time utilities

18、std::future,std::promises

19、std::async

20、随机数Random numbers

21、正则表达式regex

## 3.1、通用为本，专用为末

C++11的设计者总是希望从各种方案中抽象出更为通用的方法来构建新的特性。这意味着C++11中的新特性往往具有广泛的可用性，可以为其他已有的，或者新增的语言特性结合起来进行自由组合，或者提升已有特性的通用性。看到这些经过反复斟酌制定的新特性，并体会其“普适”的特性。

### 3.1.1、继承构造函数

类具有派生性，派生类可以自动获得基类的成员变量和接口（虚函数和纯虚函数，这里我们指的都是public派生）。不过基类的非虚函数则无法再被派生类使用了。这条规则对于类中最为特别的构造函数也不例外，如果派生类要使用基类的构造函数，通常需要在构造函数中显示声明。如：

|  |
| --- |
| struct A { A(int i) };  struct B:A { B(int i):A(i) {} }; |

B派生于A，B又在构造函数中调用A的构造函数，从而完成构造函数的”传递”。这在C++代码中非常常见。

但是，有的时候，我们的基类可能拥有数量众多的不同版本的构造函数。那么当基类中有大量的构造函数，而派生类却只有一些成员函数时，那么对于派生类而言，其构造就等同于构造基类。这个时候，在派生类中我们写的构造函数完完全全就是为了构造基类。为了语法规则，我们需要写很多的”透传”的构造函数。这无疑是相当不方便的。如：

|  |
| --- |
| struct A  {  A(int i){}  A(double d, int i){}  A(float f, int i, const char\* c){}  };  struct B:A  {  B(int i):A(i){}  B(double d, int i):A(d, i){}  B(float f, int i, const char\* c ):A(f, i ,c){}  virtual void ExtraInterface(){}  }; |

事实上，在C++中已经有了一个好用的规则，如果派生类要使用基类的成员函数，可以通过using声明。如：

|  |
| --- |
| #include <iostream>  using namespace std;  struct Base  {  void f(double i){cout << "Base:" << i << endl;}  };  struct Derived:Base  {  using Base::f;  void f(int i){cout << "Derived:"<< i<< endl;}  };  int main()  {  Base b;  b.f(4.5);// Base:4.5  Derived d;  d.f(4.5);// Base:4.5  } |

使用using声明，声明派生类Derived也可以使用基类版本的函数f。这样，派生类中实际上就拥有两个f函数的版本。

在C++11中，这个想法扩展到了构造函数上。子类可以通过使用using声明来声明继承基类的构造函数。如：

|  |
| --- |
| struct A  {  A(int i){}  A(double d, int i){}  A(float f, int i, const char\* c){}  };  struct B:A  {  using A::A;//继承构造函数  virtual void ExtraInterface(){}  }; |

C++11标准继承构造函数被设计为跟派生类中的各种类默认构造函数（默认构造、析构、拷贝构造等）一样，是隐式声明的。如果一个继承构造函数不被相关代码使用，编译器不会为其产生真正的函数代码。

不过继承构造函数只会初始化基类中成员变量，不能初始化派生类中的成员变量。

不过配合类成员的初始化表达式（C++11特性），为派生类成员变量设定一个默认值是没有问题。

|  |
| --- |
| struct A  {  A(int i){}  A(double d, int i){}  A(float f, int i, const char\* c){}  };  struct B:A  {  using A::A;//继承构造函数  int d{0};  };  int main()  {  B b(356);//b.d被初始化为0  } |

基类构造函数的参数会有默认值。对于继承构造函数来说，参数的默认值是不会被继承的。事实上，默认值会导致基类产生多个构造函数的版本，这些函数版本都会被派生类继承。必须小心使用带参数默认值的构造函数的基类。

继承构造函数”冲突”的情况。这通常发生在派生类拥有多个基类的时候。多个基类中的部分构造函数可能导致派生类中的继承构造函数的函数名、参数都相同，那么继承类中的冲突的继承构造函数将导致不合法的派生类代码。如：

|  |
| --- |
| struct A { A(int) {} };  struct B { B(int) {} };  struct C:A,B  {  using A::A;  using B::B;  } |

此时会导致C中重复定义相同类型的继承构造函数。这种情况可以显式定义继承类的冲突的构造函数，阻止隐式生成相应的继承构造函数。编译器目前不支持，编译不通过。

|  |
| --- |
| struct A { A(int) {} };  struct B { B(int) {} };  struct C:A,B  {  using A::A;  using B::B;  C(int){}  }; |

### 3.1.2、委派构造函数

与继承构造函数类似，委派函数也是C++11对C++的构造函数的一项改进，其目的也是为了减少程序员书写构造函数的时间。通过委派其他构造函数，多构造函数的类编写将更加容易。

构造函数代码冗余的例子。如

|  |
| --- |
| class Info  {  public:  Info() : type(1),name('a') { InitReset(); }  Info(int i) : type(i),name('a') { InitReset(); }  Info(char e) : type(1),name(e) { InitReset(); }  private:  void InitReset() {}  int type;  char name;  }; |

除了初始化列表有不同，其他部分基本相似，因此代码存在很多重复。

C++11中，委派构造函数是在构造函数的初始化列表位置进行构造的、委派的。

|  |
| --- |
| class Info  {  public:  Info() { InitReset(); }//目标构造函数  Info(int i) : Info() { type = i; }//委派构造函数  Info(char e) : Info() { name = e; }//委派构造函数  private:  void InitReset() {}  int type {1};  char name {'a'};  }; |

我们将一个构造函数设定为“基准版本”，比如本例的Info()版本构造函数，而其他构造函数可以通过委派“基准版本”进行初始化。

称在初始化列表中调用了“基准版本”的构造函数为委派构造函数（delegating constructor），而被调用的“基准版本”则称为目标构造函数（target constructor）。在C++11中，所谓委派构造，就是指委派函数将构造的任务委派给了目标构造函数来完成这样一种类构造的方式。

委派构造函数不能有初始化列表。在C++中，构造函数不同同时“委派”和使用初始化列表，所以如果委派函数要给变量赋初值，初始化代码必须放在函数体中。

因为初始化列表的初始化方式总是先于构造函数完成的。可以改造以下目标构造函数，使得委派构造函数依然可以在初始化列表中初始化所有成员。如：

|  |
| --- |
| class Info  {  public:  Info() : Info(1, 'a') { }  Info(int i) : Info(i, 'a') { }  Info(char e) : Info(1, e) { }  private:  void InitReset();  Info(int i, char e):type(i), name(e) { InitReset()}  int type;  char name;  }; |

定义一个私有的目标构造函数，这个构造函数接受两个参数，并将参数在初始化列表中初始化。

委托构造的一个很实际的应用就是使用构造模板函数产生目标构造函数。如：

|  |
| --- |
| class TDConstructed  {  template<typename T> TDConstructed(T first, T last):l(first, last) {}  list<int> l;  public:  TDConstructed(vector<short> & v):  TDConstructed(v.begin(), v.end()) {}  TDConstructed(deque<int> & d):  TDConstructed(d.begin(), d.end()) {}  }; |

我们定义了一个构造函数模板。而通过两个委派构造函数的委托，构造函数模板会被实例化。T分别会被推导为vector<short>::iterator和deque<int>::iterator两种类型。这样，TDConstructed类就可以很容易接受多种容器对其进行初始化。这比罗列不同类型的构造函数方便很多。

### 3.1.3、快速初始化成员变量

C++98支持在类声明中，通过“=”来初始化类中的静态成员变量。这种声明方式成为“就地”声明。但是C++98对非静态成员变量的初始化必须在构造函数中进行。

C++11标准允许非静态成员变量的初始化有多种形式。除了初始化列表外，标准还允许使用等号=或者花括号{ }进行就地的非静态成员变量的初始化。如：

|  |
| --- |
| struct init{ int a= 1; double b {1.2} }; |

### 3.1.4、右值引用：移动语义和完美转发

1、移动语义

C++程序，如果在类中包含了一个指针成员的话，要特别小心拷贝构造函数的编写，因为会出现内存泄漏。

在C++11中，这样的“偷走”临时变量中资源的构造函数，就被称为“移动构造函数“。而这样的”偷“的行为，则称之为”移动语义“（move semantics）

移动构造函数如下：

|  |
| --- |
| HasPtrMem(HasPtrMem && h):d(h.d) //移动构造函数  {  h.d = nullptr; //将临时值得指针成员置为空  } |
|  |

移动构造函数使用参数h得成员d初始化了本对象得成员d（而不是像拷贝构造函数一样需要分配内存，然后将内容依次拷贝到新分配的内存中），而h得成员d随后被置为指针空值nullptr。这就完成了移动构造得全过程。

这里所谓得“偷“堆内存，就是指将对象d指向h.d所指得内存这一条语句，相应地，我们还将h得成员d置为指针空值。在移动构造完成之后，临时对象会立即被析构。如果不改变h.d（临时对象得指针成员）的话，则临时对象会析构掉本是我们”偷“来得堆内存。

2、左值、右值与右值引用

在C++中有一个被广泛认同得说法，那就是可以取地址的、有名字得就是**左值**，反之，不能取地址的、没有名字的就是**右值**。

比如:a=b+c; &a是允许的操作，但&(b+c)这样的操作则不会通过编译。

在C++11中，右值是由两个概念构成的，一个是将亡值（xvalue, eXpiring Value），一个是纯右值(prvalue, Pure Rvalue)。

纯右值是C++98标准中右值的概念，是用于辨识临时变量和一些不跟对象关联的值。比如非引用返回的函数返回的临时变量值就是一个纯右值。一些运算表达式，比如1+3产生的临时变量值，也是纯右值。此外，类型转换函数的返回值、lambda表达式等，也都是右值。

将亡值是C++11新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用T&&的函数返回值、std::move的返回值，或者转换为T&&的类型转换函数的返回值。

在C++11中，**右值引用**就是对一个右值进行引用的类型。事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。通常情况下，我们只能是从右值表达式获得其引用。比如：

|  |
| --- |
| T && a = ReturnRvalue(); |

假设ReturnRvalue返回一个右值，我们就声明一个名为a的右值引用，其值等于ReturnRvalue函数返回的临时变量的值。

我们称C++98中的引用为“**左值引用**”。

ReturnRvalue函数返回的右值在表达式语句结束后，其生命也就终结了（具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命周期将与右值引用类型变量a的生命周期一样。

相比于以下声明方式:

|  |
| --- |
| T b = ReturnRvalue(); |

因为a是右值引用，直接绑定了ReturnRvalue()返回的临时量，而b只是由临时量构造而成，而临时量在表达式结束后会析构，因此会多一次析构和构造的开销。

问题：

我使用以下代码验证右值引用，结果不像预期的那样，它们都是构造一次、析构依次，并没有多一次析构和构造的过程。不知道哪里出了问题，这个问题暂时留在这里。

|  |
| --- |
| #include <iostream>  using namespace std;  class T  {  public:  T():m(0)  {  cout << "T construct :" << m << endl;  }  T(int i):m(i)  {  cout << "T construct with para :" << m << endl;  }  T & operator=(const T & rhs)  {  cout << "T assign construct :" << m << endl;  }  ~T()  {  cout << "T destruct :" << m << endl;  }    private:  int m;  };  T ReturnRvalue()  {  T t(9);  return t;  }  int main()  {  //c++98  T a = ReturnRvalue();  T b = a;    //c++11  //T && b = ReturnRvalue();  } |

### 3.1.5、std::move 强制转化为右值

std::move唯一的功能是将一个左值强制转化为右值引用，继而我们可以通过右值引用使用该值，以用于移动语义。从实现上讲，std::move基本等同于一个类型转换：

|  |
| --- |
| static\_cast<T&&>(lvalue); |

被转化的左值，其生命期并没有随着左右值得转化而改变。

为了保证移动语义的传递，程序员在编写移动构造函数的时候，应该总是记得使用std::move转换拥有形如堆内存、文件句柄等资源的成员为右值，这样，如果成员支持移动构造的话，就可以实现其移动语义。而即使成员没有移动构造函数，那么接受常量左值的构造函数版本也会轻轻松松地实现拷贝构造，因此不会引起大的问题。

### 3.1.6、移动语义补充

移动语义一定是要修改临时变量的值。

如果声明移动构造函数：

|  |
| --- |
| Moveable(const Moveable &&)  const Moveable ReturnVal(); |

都会使得临时变量常量化，成为一个常量右值，那么临时变量的引用也就无法修改，从而导致无法实现移动语义。

在C++11中，拷贝/移动构造函数实际上有以下3个版本：

|  |
| --- |
| T object(T &)  T object(const T &)  T object(T &&) |

其中常量左值引用（第2个）的版本是一个拷贝构造版本，而右值引用版本（第3个）是一个移动构造版本。

只实现一种语义在类的编写中也是非常常见。而只有移动语义的类型时，因为只有移动语义表明该类型的变量所拥有的资源只能被移动，而不能被拷贝。那么这样的资源必须是唯一的。因此，只有移动语义构造的类型往往都是“资源型”的类型，比如说智能指针，文件流等，都可以视为“资源型”的类型。

### 3.1.7、完美转发

perfect forwarding是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另外一个函数。比如：

|  |
| --- |
| template <typename T>  void IamForwarding(T t) { IrunCodeActually(t); } |

IamForwarding是一个转发函数模板。而函数IrunCodeActually则是真正执行代码的目标函数。

对于目标函数IrunCodeActually，它总是希望转发函数将参数按照传入IamForwarding时的类型传递，而不产生额外的开销，就好像转发者不存在一样。

通常，使用引用类型，引用类型不会有拷贝的开销。其次，则需要考虑转发函数对类型的接受能力。因为目标函数可能需要能够即接受左值引用，又接受右值引用。

C++11引入了一条“引用折叠”（reference collapsing）的新语言规则，并结合新的模板推导规则来完成完美转发。

|  |
| --- |
| typedef const int T;  typedef T& TR;  TR& v = 1; |

TR& v = 1;在C++98中不合法，但在C++11中，一旦出现这样的表达式，就会发生引用折叠，即将复杂的位置表达式折叠为已知的简单表达式。



一旦定义中出现了左值引用，引用折叠总是优先将其折叠为左值引用。

# 四、深入应用C++11：代码优化与工程级应用

In-Depth C++11 Code Optimization and Engineering Level Application

# 参考文献

1、《深入应用C++11 代码优化与工程级应用》，祈宇，机械工业出版社；

2、<https://stackoverflow.com/questions/1518711/how-does-free-know-how-much-to-free>

3、<http://gaoshen.site/2013-5-20-aboutfree/>

4、深入理解C++11：C++11新特性解析与应用，IBM XL编译器中国开发团队，机械工业出版社；