1. 基本数据类型：

|  |
| --- |
| void, bool, char, short  Int, float, double, long |

一个char的空间应确保可以存放机器基本字符集中任意字符对应的数字值，也就是说，一个char的大小和一个机器字节一样。

类型char的表现形式有可能是unsinged char，也有可能是signed char，具体是哪种由编译器决定。

1. 变量定义

**基本数据类型** (一组)**类型声明符**

面对一条比较复杂的指针或引用的声明语句时，从右向左阅读有助于弄清楚它的真实含义。

默认情况下，类型修饰符从右向左依次绑定。

但是，如果是数组的声明，最好的办法是从数组的名字开始，由内而外(有小括号的情况)，再由右向左的顺序去阅读。

1. 变量初始化

定义于任何函数体之外的内置类型变量被初始化为0；函数体内的内置类型变量如果没有初始化，则其值未定义。

类对象如果没有显示地初始化，则其值由类确定。

建议初始化每一个内置类型的变量。

1. Void\* 指针

Void\* 指针能做的事儿比较有限：拿它和别的指针比较，作为函数的输入或者输出，赋值给另外一个void\* 指针。

以void\* 的视角来看内存空间就仅仅是内存空间，无法访问内存空间中所存的对象。

1. 常量指针、常量引用

所谓指向常量的指针或引用，不过是指针或引用“自以为是”罢了，它们觉得自己指向了常量，所以自觉地不去改变指向对象的值。

1. constexpr和常量表达式

常量表达式是指值不会改变并且在编译过程中就能得到计算结果的表达式。显然，字面值属于常量表达式，用常量表达式初始化的const对象也是常量表达式。

C++11新标准规定，允许将变量声明为constexpr类型，以便由编译器来验证变量的值是否是一个常量表达式。声明为constexpr的变量一定是一个常量，而且必须用常量表达式初始化：

constexpr int mf = 20;

constexpr int limit = mf + 1;

新标准还允许定义一种特殊的constexpr函数，这种函数应该能够足够简单以使得编译时就可以计算其结果，这样就能使用constexpr函数去初始化constexpr变量了。

一般来说，如果你认定变量是一个常量表达式，那就应该把它声明成constexpr类型。

1. 类型别名

有两种方法用于定义类型别名：

1. 传统方法，使用关键字**typedef**:

typedef double wages: //wage是double的同义词

1. C++11的新方法，使用别名声明(alias declaration)来定义：

using wages = double;//wage是double的同义词

如果某个类型别名指代的是复合类型或者常量，则把它用到声明语句中就会产生意想不到的后果：

typedef char \*pstring;

const pstring cstr=0; //**等价于char \* const cstr=0;不是const char \*cstr=0;**

1. auto类型说明符

C++11新标准引入了auto类型说明符，和原来那些只对应一种特定类型的说明符(比如int, double)不懂，auto让编译器通过初始值来推算变量的类型，显然，auto定义的变量必须有初始化值：

auto item = val1 + val2; //item初始化为val1和val2相加的结果

1. decltype类型指示符

希望从表达式的类型推断出要定义的变量的类型，但不想用该表达式的值初始化变量。C++11引入decltype，它的作用是选择并返回操作数的数据类型。编译器分析表达式并得到它的类型，却不实际计算表达式的值。

|  |
| --- |
| for(decltype(vec.size()) i= 0； i < vec.size(); ++i)  cout<<vec[i]<<endl;  for(decltype(s.size()) idx = 0; idx<s.size() && !isspace(s[idx]); ++idx)  s[idx] = toupper(s[idx]); |

切记：decltype((variable))（注意，是双层括号）的结构永远是引用，而decltype(variable)结果只有当variable本身就是一个引用时才是引用。

1. 自定义数据结构struct

struct Sales\_data{}

string bookNo;

unsigned units\_sold = 0;

double revenue = 0.0;  
}；

C++11新标准规定，可以为数据成员提供一个类内初始化值(in-class initializer)，创建对象时，类内初始化值将用于初始化数据成员，没有初始化值得成员将被默认初始化。

1. 范围for(range for)语句

这种语句遍历给定序列中的每个元素，并对序列中的每个值执行某种操作，其语法形式为：

for (declaration: expression)

statement

|  |
| --- |
| string str(“some string”)  //每行输出str中的一个字符  for(auto c: str)  cout<< c << endl; |

如果想要在上诉for语句中改变expression，则必须把declaration定义成引用类型形式。记住，所谓引用只是给定对象的一个别名，因此当使用引用作为循环控制变量时，这个变量实际上依次被绑定到序列上的每个元素。

|  |
| --- |
| string s(“Hello World!”)  //转换成大写形式  For(auto &c : s)  c = toupper(c) |

1. vector

C++标准要求vector应该能在运行时高效快速地添加元素，因此，为了vector对象能高效地增长，那么在定义vector对象的时候设定其大小就没有必要了。只有一种情况例外，那就是所有元素的值都一样。

范围for语句体内不应该改变其所遍历序列的大小(线程不安全的证据？)。

1. 数组

数组是一种复合类型。数组中元素的个数也属于数组类型的一部分，编译的时候纬度应该是已知的。也就是说，数组纬度必须是一个常量表达式。

**现代的C++程序员应当尽量使用vector和迭代器，避免使用内置数组和指针；应该尽量使用string，避免使用C风格的基于数组的字符串。**

1. 强制类型转换

**static\_cast:**

任何具有明确定义的类型转换，**只要不包含底层const，都可以使用static\_cast**。当需要把一个较大的算术类型赋值给较小的类型时，static\_cast非常有用。此时，强制类型转换告诉程序的读者和编译器：我们知道并且不在乎潜在的精度损失。

static\_cast对于编译器无法自动执行的类型转换也非常有用，比如可以使用staitc\_cast找回存在于void\*指针中的值。

**const\_cast:**

const\_cast只能改变运算对象的底层const。一旦我们去掉某个对象的const性质，编译器就不再阻止我们对该对象进行写操作了。如果对象本身不是一个常量，使用强制类型转换获得写权限是合法的行为。如果对象是一个常量，再使用const\_cast执行写操作就会产生未定义的后果。

const\_cast常用于有函数重载的上下文中。使用const\_cast的目的,在于某些变量原本不是const的,但由于某种特殊原因,无意间被变成了const的。例如使用了一个const引用指向了一个本来不是const的对象，结果写了一些代码之后发现它实际上需要被修改。这在平时的工作中不会遇到，因为你可以直接把const引用修改成非const的，但C++中可能的情况太多，尤其考虑到很多复用的时候，有时还是会出现本不该是const的对象被const引用了这种情况。尤其是使用模板，比较复杂的情况。

**reinterpret\_cast:**

reinterpret\_cast通常为运算对象的位模式提供较低层次上的重新解释，本质上依赖于机器，要想安全使用reinterpret\_cast必须对涉及的类型和编译器实现转换的过程都非常了解。

允许将任何指针转换为任何其他指针类型。 也允许将任何整数类型转换为任何指针类型以及反向转换。reinterpret\_cast运算符是用来处理无关类型之间的转换；它会产生一个新的值，这个值会有与原始参数（expressoin）有完全相同的比特位。

所以总结来说：reinterpret\_cast用在**任意指针（或引用）类型之间的转换**；以及**指针与足够大的整数类型之间的转换**；从**整数类型（包括枚举类型）到指针类型，无视大小**。

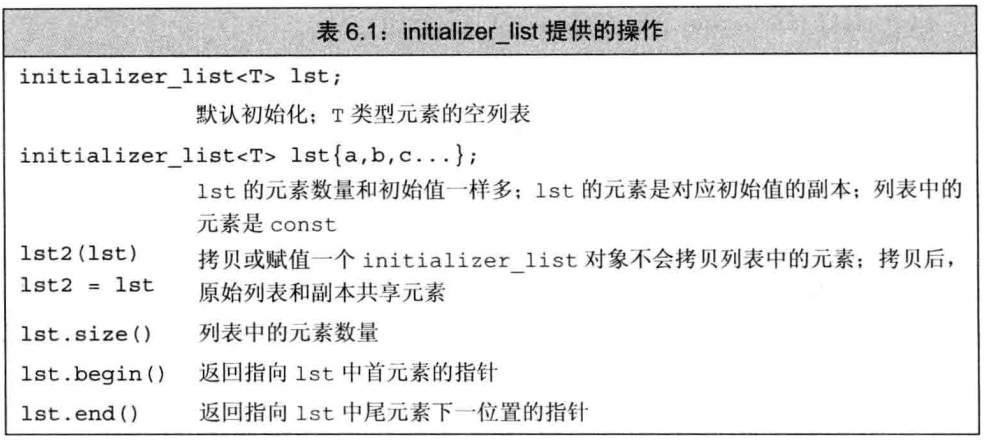
reinterpret\_cast不能像const\_cast那样去除const修饰符。

**dynamic\_cast:**

1. 含有可变形参的函数

为了能编写能处理不同数量实参的函数，C++11新标准提供了两种主要的方法：如果所有的实参类型相同，则可以传递一个名为initializer\_list的标准库类型；如果实参的类型不同，我们可以编写一种特殊的函数，也就是所谓的可变参数模板。

C++还提供了一种特殊的形参类型----省略符形参，可以用它传递可变数量的实参，这种功能一般只用于与C函数交互的接口程序。



|  |
| --- |
| Void error\_msg(ErrorCode e, initializer\_list<string> il)  {  Cout<<e.msg()<<endl;  For(const auto &elem: il)  Cout<<elem<<” “;  Cout<<endl;  }  Void error\_msg2(initializer\_list<string> il)  {  For(auto beg=il.begin(); beg != il.end(); ++beg)  Cout<<\*beg<<””;  Cout<<endl;  }  …  If(excepted != actual)  Error\_msg(ErrorCode(42), {“functionX”,expected, actual});  Else  Error\_msg(ErrorCode(0),{“functionX”,”okay”});  … |

省略符形参：

省略符形参是为了便于C++程序访问某些特殊的C代码而设置的，这些代码使用了名为varargs的C标准库功能，通常情况下省略符形参不应用于其他目的。

省略符形参只能出现在形参列表的最后一个位置，它的形式无外乎以下两种：

void foo(param\_list, …);

void foo(…);

|  |
| --- |
| 我们使用可变参数应该有以下步骤:  ⑴在程序中将用到以下这些宏:  void va\_start( va\_list arg\_ptr, prev\_param );  type va\_arg( va\_list arg\_ptr, type );  void va\_end( va\_list arg\_ptr );  va在这里是variable-argument(可变参数)的意思.  这些宏定义在stdarg.h中,所以用到可变参数的程序应该包含这个头文件. |