

# C++\_Basic 总结

郑华

2018 年 12 月 28 日



# 目录

<b>第一章 C++11 新特性必学</b>	<b>11</b>
1.1 右值操作	11
1.1.1 右值与左值	11
1.1.2 右值引用	11
1.1.3 Move 语义	12
简介	12
使用场景	14
STL 使用	15
1.1.4 完美转发	17
1.2 面向对象改进	18
1.2.1 override 与 final 标识符	18
1.2.2 成员初始化	19
1.2.3 Defaulted	20
-引入	20
-使用	21
1.2.4 Deleted	23
-引入	23
-使用	24
1.2.5 委托构造	25
1.3 类型改进	26
1.3.1 类型推导	26
-auto	26
-decltype	26
1.3.2 类型别名	26
1.3.3 nullptr	27
1.3.4 constexpr	27
1.3.5 R-raw string	27
1.3.6 Uniform Initialization	28
1.3.7 Initialize List	28
1.3.8 Explicit Conversion Operators	29
1.3.9 Smart Pointers	29

shared_ptr . . . . .	29
unique_ptr . . . . .	30
weak_ptr . . . . .	31
1.4 处理方式 (函数) 改进 . . . . .	32
1.4.1 匿名函数 (也叫 Lambda 表达式) . . . . .	32
1.4.2 for 的变革 . . . . .	33
1.4.3 非成员 begin() 和 end() . . . . .	33
1.4.4 静态断言 . . . . .	34
1.5 参考 . . . . .	35
<b>第二章 return 语句</b>	<b>37</b>
2.1 传值返回 . . . . .	37
2.2 引用返回 . . . . .	37
例子 . . . . .	37
<b>第三章 inline 关键字</b>	<b>39</b>
3.1 使用 . . . . .	39
在函数定义中函数返回类型前加上关键字 inline, 注: 声明不起作用 . . . . .	39
定义在类声明之中的成员函数将自动地成为内联函数 . . . . .	39
3.2 效率 . . . . .	39
3.2.1 实现机制 . . . . .	39
3.2.2 慎用内联 . . . . .	40
3.3 内联技巧 . . . . .	40
3.3.1 条件内联 . . . . .	40
3.3.2 选择性内联 . . . . .	40
3.4 结论 . . . . .	41
<b>第四章 重载</b>	<b>43</b>
4.1 函数重载 . . . . .	43
4.2 函数重载后的隐藏规则 . . . . .	43
4.3 运算符重载 . . . . .	44
4.3.1 重载 = 运算符 . . . . .	44
4.3.2 重载 'output' 运算符 . . . . .	44
4.3.3 重载 New Delete 运算符 . . . . .	44
4.3.4 重载 ++ 运算符 . . . . .	44
4.4 函数的默认参数 . . . . .	44
<b>第五章 struct And union</b>	<b>47</b>
5.1 Struct . . . . .	47
5.1.1 定义 . . . . .	47
5.1.2 函数使用 . . . . .	47

传值 . . . . .	47
初始化 . . . . .	48
5.1.3 变长结构体 . . . . .	48
注意 . . . . .	49
5.2 Union . . . . .	49
5.2.1 定义 . . . . .	49
5.3 字节对齐-StructAndUnion . . . . .	50
<b>第六章 string</b>	<b>51</b>
6.1 方法 . . . . .	51
6.1.1 string 类的构造函数 . . . . .	51
6.1.2 string 类的字符操作 . . . . .	52
6.2 操作符 . . . . .	53
6.2.1 += 操作符 . . . . .	53
6.2.2 + 操作符 . . . . .	53
6.2.3 == 操作符 . . . . .	53
6.3 宽字符 wchar_t . . . . .	53
<b>第七章 class</b>	<b>55</b>
7.1 C++ 中使用 Struct 创建 1 个类 . . . . .	55
7.2 C++ 中使用 Class 创建 1 个类 . . . . .	55
7.3 const 型成员函数 . . . . .	55
7.4 析构函数与构造函数 . . . . .	55
7.4.1 潜规则 . . . . .	55
7.4.2 Initializer-List Constructors . . . . .	56
7.4.3 初始化顺序 . . . . .	57
7.5 值语义与对象语义 . . . . .	57
7.5.1 值语义 . . . . .	57
7.5.2 对象语义 . . . . .	57
7.6 拷贝构造函数 . . . . .	57
7.7 初始化列表 . . . . .	58
7.8 一些只有 . . . . .	58
7.9 STATIC 修饰词 . . . . .	59
7.9.1 初始化时间 . . . . .	59
7.9.2 静态全局变量 . . . . .	59
7.9.3 静态局部变量 . . . . .	60
7.9.4 静态函数 . . . . .	60
7.9.5 静态数据成员 . . . . .	60
7.9.6 静态成员函数 . . . . .	60

<b>第八章 this</b>	<b>63</b>
8.1 概念	63
8.2 注意问题	63
<b>第九章 继承</b>	<b>65</b>
9.1 潜规则	65
9.2 构造函数调用顺序	67
9.3 继承方式	67
9.3.1 继承后对父类资源的访问权限	67
9.3.2 继承约束	67
禁止继承-final	67
9.3.3 重写父类方法	67
9.3.4 切片	68
9.3.5 使用父类方法	69
9.4 支持多继承	69
9.5 虚基类	69
9.6 C++ 接口的实现方式	72
9.6.1 纯虚函数	72
9.6.2 抽象类	72
9.6.3 C++ 接口与抽象类的区别	72
<b>第十章 virtual 关键字</b>	<b>73</b>
10.1 多态	73
10.2 虚函数表	74
10.2.1 一般继承（无虚函数覆盖）	76
10.2.2 一般继承（有虚函数覆盖）	77
<b>第十一章 C++ 内存结构</b>	<b>79</b>
11.1 进程结构	79
11.1.1 Stack 栈	79
11.1.2 Heap 堆	79
11.1.3 .bss 未初始化全局与静态变量	80
11.1.4 .data 初始化的全局与静态变量	80
11.1.5 .text 代码区	80
11.2 对象结构	80
11.2.1 虚函数在内存中分布	80
11.2.2 单一对象	81
11.2.3 单一对象 (virtual)	82
11.2.4 单继承 (base 无 virtual)	83
11.2.5 单继承 (base 有 virtual)	84

11.2.6 虚继承-单继承 (virtual) . . . . .	85
11.2.7 多继承 (base 有 virtual) . . . . .	86
11.2.8 虚继承-多继承 (base 有 virtual) . . . . .	89
11.3 ELF . . . . .	91
<b>第十二章 extern 关键字</b>	<b>93</b>
12.1 基本解释 . . . . .	93
12.2 问题 . . . . .	93
作用范围-可见性的关键字 . . . . .	93
连接申明 linkage declaration . . . . .	93
<b>第十三章 volatile 关键字</b>	<b>95</b>
13.1 含义 . . . . .	95
13.2 示例 . . . . .	95
声明时语法 . . . . .	95
volatile 变量调用过程 . . . . .	95
13.3 volatile 指针 . . . . .	97
13.4 多线程下的 volatile . . . . .	97
13.5 参考 . . . . .	98
<b>第十四章 explicit 关键字</b>	<b>99</b>
14.1 作用 . . . . .	99
<b>第十五章 friend 关键字</b>	<b>101</b>
15.1 目的 . . . . .	101
15.2 友元函数 . . . . .	101
15.3 友元类 . . . . .	102
<b>第十六章 mutable 关键字</b>	<b>105</b>
16.1 目的 . . . . .	105
16.2 示例 . . . . .	105
<b>第十七章 类型转换</b>	<b>107</b>
17.1 static_cast . . . . .	107
17.2 const_cast . . . . .	107
17.3 dynamic_cast . . . . .	107
17.4 reinterpret_cast . . . . .	108
<b>第十八章 static_assert 关键字</b>	<b>111</b>
18.1 简介 . . . . .	111
18.2 说明 . . . . .	111
18.3 范例 . . . . .	111

18.4 参考 . . . . .	112
<b>第十九章 命名空间</b>	<b>113</b>
19.1 namespace 别名 . . . . .	113
19.2 namespace 扩充 . . . . .	113
<b>第二十章 引用</b>	<b>115</b>
20.1 引用创建时必须初始化 . . . . .	115
20.2 引用不能重定义 . . . . .	115
20.3 函数中的引用 . . . . .	115
20.4 引用右值 . . . . .	116
<b>第二十一章 enum 的使用</b>	<b>119</b>
21.1 定义 . . . . .	119
21.2 使用 . . . . .	119
21.2.1 类外函数 . . . . .	119
21.2.2 类中 . . . . .	119
21.3 注意事项 . . . . .	120
<b>第二十二章 模版</b>	<b>121</b>
<b>第二十三章 异常</b>	<b>123</b>
<b>第二十四章 编译</b>	<b>125</b>
24.1 Windows-VisualStudio . . . . .	125
24.1.1 配置全局 VC++ 目录 . . . . .	125
24.1.2 配置第三程序库-以 opencv 示例 . . . . .	125
下载配置所需 . . . . .	125
安装 . . . . .	125
配置 . . . . .	125
编程测试 . . . . .	126
24.1.3 dll 与 lib 的配置 . . . . .	126
24.1.4 常见错误 . . . . .	126
. . . . .	126
24.2 Linux - Makefile . . . . .	126
24.2.1 Makefile 规则 . . . . .	127
示例 . . . . .	127
24.2.2 make 执行流程 . . . . .	127
24.2.3 makefile 中使用变量 . . . . .	128
24.2.4 make 的自动推倒 . . . . .	128
24.2.5 添加依赖库 . . . . .	129



24.2.6 代码泄漏检测 . . . . .	129
24.2.7 参考文献 . . . . .	129
<b>第二十五章 调试</b>	<b>131</b>
25.1 Linux - 调试 GDB . . . . .	131
25.1.1 使用 GDB . . . . .	131
-g 编译项目 . . . . .	131
启动 GDB . . . . .	131
GDB 常用命令 . . . . .	131
25.1.2 参考 . . . . .	133
<b>第二十六章 工程-细节</b>	<b>135</b>
26.1 语言使用基本问题 . . . . .	135
26.1.1 注释格式 . . . . .	135
函数注释 . . . . .	135
代码注释 . . . . .	135
26.1.2 预编译 . . . . .	136
26.1.3 变量问题 . . . . .	137
变量 (包括成员) 命名 . . . . .	137
使用 . . . . .	137
26.1.4 运算符使用问题 . . . . .	137
26.1.5 函数问题 . . . . .	137
26.1.6 条件语句问题 . . . . .	137
26.1.7 循环语句问题 . . . . .	138
26.1.8 数值类型转换问题 . . . . .	138
26.2 内存管理 . . . . .	138
26.2.1 内存分配与使用 . . . . .	138
26.2.2 内存泄漏 . . . . .	138
26.3 缓冲区溢出 . . . . .	138
26.3.1 数组越界 . . . . .	138
26.3.2 数据越界 . . . . .	138
26.3.3 字符串操作溢出 . . . . .	138
26.4 指针问题 . . . . .	138
26.4.1 定义问题 . . . . .	138
26.4.2 空指针解引用 . . . . .	139
26.4.3 指针非法使用 . . . . .	139
26.4.4 数组参数 . . . . .	139
26.5 安全缺陷 . . . . .	139
26.5.1 外部输入安全缺陷 . . . . .	139
26.5.2 资源泄漏 . . . . .	139

26.5.3	其他	139
26.6	类	139
26.6.1	类命名问题	139
26.6.2	访问权限	140
26.6.3	区分对象所在处	140
	Object on the Stack	140
	Object on the Heap	140
26.7	其他	140
26.7.1	预处理	140
26.7.2	异常	140
26.7.3	多线程和同步性	141
26.7.4	代码不可达	141
26.7.5	关于 VS2013 中的相对路径问题	141
26.7.6	编译	141

# 第一章 C++11 新特性必学

## 1.1 右值操作

### 1.1.1 右值与左值

- **左值**：左值是一个可以用来存储数据的变量，有实际的内存地址，表达式结束后依然存在。她因在赋值操作符左边而得名。
- **右值**：更准确的应该叫非左值，是一个匿名的临时变量，她在表达式结束时生命周期终止，不能存放数据，可以被修改，也可以不被修改（若被 `const` 标识）

根据这个解释，鉴别左值和右值最简单的方法是：左值可以用取地址符号“&”获取地址，而右值无法使用“&”（会编译错误）。

```
int x = 0; //对象实例，有名，x 为左值
int* p = &++x; //可以取地址，++x 是左值
++x = 10; //前置++ 返回的是左值，可以赋值
p = &x++; //后置++ 返回一个临时对象，不能取地址或赋值，是右值，编译错误
```

因为右值是临时的，生命周期即将结束，之后无人关心他的值，所以我们可以把他的所有内容转移到其他对象中，从而完全消除高昂的拷贝代价。

### 1.1.2 右值引用

**表示：** 有了右值的概念，右值引用应运而生。C++ 使用 `T&&` 表示右值引用，而原来的 `T&` 表示左值引用，分别引用右值对象和左值对象。

**核心：** 对一个对象使用右值引用，意味着显式的标记这个对象为右值，可以被转移来优化。同时也相当于为他添加了一个“临时的名字”，生命周期得到了延长，不会在表达式结束时消失，而是与右值引用绑定在一起。

其实就是存储临时对象的地址。

**右值引用绑定的对象** 返回非引用类型的函数，产生右值的表达式（算术表达式、关系表达式、位、后置递增递减）

生命周期 `int && num = 8; //右值引用`

正常情况下，右值”8“在表达式语句结束后，其生命也就终结了（通常我们也称其具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命期将与右值引用类型变量 `num` 的生命期一样。只要 `num` 还“活着”，该右值临时量将会一直“存活”下去。

示例：

```
int& r1 = ++x; //左值引用
int&& r2 = x++; //右值引用，引用了自增后的临时对象 Xvalue
const int& r3 = x++; //常量左值引用
const int&& r4 = x++; //常量右值引用，不能修改，不能转移，无实际意义
cout << r2 << endl; //右引用延长生命期，右值对象在表达式结束后仍然存在
```

<https://github.com/ctzhenghua/C-NetworkPractice-Code/blob/Dev/C%2B%2B/Basic/RightValue/RightReference.cc>

### 1.1.3 Move 语义

简介

`move` 是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝，如图所示是深拷贝和`move` 的区别

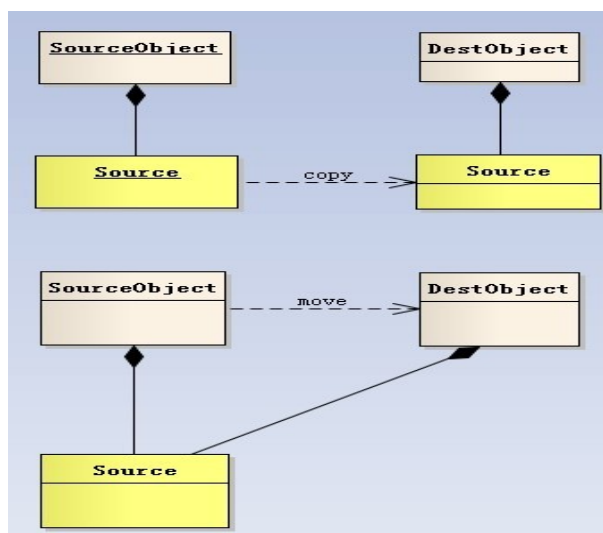


图 1.1: UML 图

```
//执行深拷贝
String(const String& rhs): data_(new char[rhs.size() + 1])
{
    strcpy(data_, rhs.c_str());
}
```

这里进行了内存分配和拷贝数据。

如果rhs 是个临时对象，要是能将 rhs 的数据“move”到data\_ 中岂不是提高了运行效率，这样子你即不需要为data\_ 重新分配内存，又不需要去释放rhs 的内存，简直两全其美。

```
String(String&& rhs): data_(rhs.data_)
{
    rhs.data_ = nullptr;
}
```

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。如果vector 没有 move 构造函数，那么复制构造函数将被调用，这时进行深拷贝。

细节对比如下：

```
//////////////////////拷贝构造//////////////////////
class ArrayWrapper
{
public:
    ArrayWrapper (int n): _p_vals( new int[ n ] ), _size( n){}
    // copy constructor
    ArrayWrapper (const ArrayWrapper& other): _p_vals( new int[ other._size ] ), _size
        ( other._size )
    {
        for ( int i = 0; i < _size; ++i )
        {
            _p_vals[ i ] = other._p_vals[ i ];
        }
    }
    ~ArrayWrapper ()
    {
        delete [] _p_vals;
    }
private:
    int *_p_vals;
    int _size;
};

//////////////////////move 构造//////////////////////
class ArrayWrapper
{
public:
    // default constructor produces a moderately sized array
    ArrayWrapper () : _p_vals( new int[ 64 ] ), _size( 64 ){}

    ArrayWrapper (int n): _p_vals( new int[ n ] ), _size( n){}
```

```

// move constructor
ArrayWrapper (ArrayWrapper&& other): _p_vals( other._p_vals ), _size( other._size
    )
{
    other._p_vals = NULL;
}

//Move other
ArrayWrapper(ArrayWrapper&& other)
{
    std::swap(_p_vals,other._p_vals);
    std::swap(_size, other._size);
}

// 转移赋值函数
ArrayWrapper& operator= (ArrayWrapper&& other)
{
    std::swap(_p_vals,other._p_vals);
    std::swap(_size, other._size);
    return *this;
}

// copy constructor
ArrayWrapper (const ArrayWrapper& other): _p_vals( new int[ other._size ] ), _size
    ( other._size )
{
    for ( int i = 0; i < _size; ++i )
    {
        _p_vals[ i ] = other._p_vals[ i ];
    }
}

~ArrayWrapper ()
{
    delete [] _p_vals;
}

private:
    int *_p_vals;
    int _size;
};

```

## 使用场景

1. C++ 标准库使用比如`vector::push_back` 等这类函数时, 会对参数的对象进行复制, 连数据也会复制. 这就会造成对象内存的额外创建, 本来原意是想把参数`push_back` 进去就行了.

2. C++11 提供了 `std::move` 函数来把左值转换为 rvalue, 而且新版的 `push_back` 也支持 `&&` 参数的重载版本, 这时候就可以高效率的使用内存了
3. 对指针类型的标准库对象并不需要这么做.

<http://blog.csdn.net/infoworld/article/details/50736633>

```
void TestSTLObject()
{
    std::string str = "Hello";
    std::vector<std::string> v;

    // uses the push_back(const T&) overload, which means
    // we'll incur the cost of copying str
    v.push_back(str);
    std::cout << "After copy, str is \"\" << str << "\"\n";

    // uses the rvalue reference push_back(T&&) overload,
    // which means no strings will be copied; instead, the contents
    // of str will be moved into the vector. This is less
    // expensive, but also means str might now be empty.
    v.push_back(std::move(str));
    std::cout << "After move, str is \"\" << str << "\"\n";

    std::cout << "The contents of the vector are \"\" << v[0]
    << "\", \"\" << v[1] << "\"\n";
}

/*
After copy, str is "Hello"
After move, str is ""
The contents of the vector are "Hello", "Hello"
*/
```

如果不用 `std::move`, 拷贝的代价很大, 性能较低。使用 `move` 几乎没有任何代价, 只是转换了资源的所有权。如果一个对象内部有较大的对内存或者动态数组时, 很有必要写 `move` 语义的拷贝构造函数和赋值函数, 避免无谓的深拷贝, 以提高性能。

```
std::list< std::string > tokens; //省略初始化...
std::list< std::string > t = tokens;

std::list< std::string > tokens;
std::list< std::string > t = std::move(tokens);
```

## STL 使用

**转移语义** C++ 标准库中的 `string`、`vector`、`deque` 等组件都实现了转移构造函数和转移赋值函数, 可以利用转移语义优化, 所以现在在函数里返回一个大容器对象是非常有效的。

这些标准容器 (除std::array) 还特别的增加了emplace() 系列函数, 可以使用转移语义直接插入元素, 进一步提高运行效能。

```
vector<complex<double>> v; //标准序列容器
v.emplace_back(3,4); //直接使用右值插入元素, 无须构造再拷贝

map<string, string>m;
m.emplace("metroid","prime");//直接使用右值插入元素, 无须构造再拷贝
m.push_back("methroid","prime");// 这样是不行的, wrong
```

## 新容器

- **array**: 提供了定长数组容器array, 相比于普通数组更安全、更易使用。array 是定长数组, 所以不支持诸如插入、删除等改变容器大小的操作, 但是可以对元素进行赋值改变其值。

```
array<int, 5> c4 = {0, 1, 2, 3, 4};
c4[3] = 100; // can't insert since the array size is fixed.
for(auto it4_1 = c4.begin(); it4_1 != c4.end(); it4_1++)
{
    cout<<*it4_1<<'\t';
}
```

- **forward\_list**: 提供了一个快速的、安全的单向链表实现。因为是单向链表, 所以也就没有rbegin、rend 一类的函数支持了。

同样是因为单向链表的缘故, 无法访问到给定元素的前驱, 所以没有提供insert 函数, 而对应提供了一个insert\_after 函数, 用于在给定元素之后插入节点.erase\_after、emplace\_after 同理。

```
forward_list<int> c5 = {3, 4};
c5.push_front(2);
c5.push_front(1);
auto it5_1 = c5.before_begin();
c5.insert_after(it5_1, 0);
```

- **无序序列容器**: 引入了对map、set 等关联容器的无序版本, 叫做unordered\_map/unordered\_set。

无序关联容器不使用键值的比较操作来组织元素顺序, 而是使用哈希。这样在某些元素顺序不重要的情况下, 效率更高。

```
unordered_map<string, int> c12;
map<string, int> c13;
string string_keys[5] = {"aaa", "bbb", "ccc", "ddd", "eee"};
for(int i = 0; i < 5; i++)
{
    c12[string_keys[i]] = i;
    c13[string_keys[i]] = i;
}
```



```

cout<<"normal_map:\n";
for(auto it13 = c13.begin(); it13 != c13.end(); it13++)
    cout<<it13->first<<':'<<it13->second<<'\t';
cout<<endl;

cout<<"unordered_map:\n";
for(auto it12 = c12.begin(); it12 != c12.end(); it12++)
    cout<<it12->first<<':'<<it12->second<<'\t';
cout<<endl;

```

- **tuple**: 用于方便的将不同类型的值组合起来。

可以通过如下方式，获取tuple 中的元素、tuple 的长度等：

```

//tuple<int, string, vector<int>> c14 = {1, "tuple", {0, 1, 2, 3, 4}}; // wrong.
    must explicit initialize

tuple<int, string, vector<int>> c14{1, "tuple", {0, 1, 2, 3, 4}};

get<0>(c14) = 2; // 赋值-修改值
typedef decltype(c14) ctype;
size_t sz = tuple_size<ctype>::value;
cout<<get<0>(c14)<<'\t'<<get<1>(c14)<<'\t'<<get<2>(c14)[0]<<'\t'<<sz<<endl;

```

- **shrink\_to\_fit**: 一般可变长容器会预先多分配一部分内存出来，以备在后续增加元素时，不用每次都申请内存。所以有 size 和 capacity 之分。size 是当前容器中存有元素的个数，而 capacity 则是在不重新申请内存的情况下，当前可存放元素的最大数目。而 shrink\_to\_fit 就表示将 capacity 中的多余部分退回，使其回到 size 大小。但是，这个函数的具体效果要依赖于编译器的实现

```

vector<int> c11;
for(int i = 0; i < 24; i++)
    c11.push_back(i);

cout<<c11.size()<<'\t'<<c11.capacity()<<endl;
c11.shrink_to_fit();
cout<<c11.size()<<'\t'<<c11.capacity()<<endl;

```

### 1.1.4 完美转发

右值引用类型是独立于值的，一个右值引用参数作为函数的形参，在函数内部再转发该参数的时候它已经变成一个左值，并不是他原来的类型。

如果我们需要一种方法能够按照参数原来的类型转发到另一个函数，这种转发类型称为完美转发

```

template<typename T>
void print(T& t){
    cout << "lvalue" << endl;
}
template<typename T>
void print(T&& t){
    cout << "rvalue" << endl;
}

template<typename T>
void TestForward(T && v){
    print(v);
    print(std::forward<T>(v));
    print(std::move(v));
}

int main(){
    TestForward(1); // l r r
    int x = 1;
    TestForward(x); // l l r
    TestForward(std::forward<int>(x)); // l r r
    return 0;
}

```

## 1.2 面向对象改进

### 1.2.1 override 与 final 标识符

我总觉得 C++ 中虚函数的设计很差劲，因为时至今日仍然没有一个强制的机制来标识虚函数会在派生类里被改写。**virtual** 关键字是可选的，这使得阅读代码变得很费劲。因为可能需要追溯到继承体系的源头才能确定某个方法是否是虚函数。为了增加可读性，我总是在派生类里也写上 **virtual** 关键字，并且也鼓励大家都这么做。即使这样，仍然会产生一些微妙的错误。看下面这个例子：

```

class A
{
public:
    virtual void f(short) {std::cout << "A::f" << std::endl;}
};
class B : public A
{
public:
    virtual void f(int) {std::cout << "B::f" << std::endl;}
};

```

**B::f** 按理应当重写 **A::f**。然而二者的声明是不同的，一个参数是 `short`，另一个是 `int`。因此 **B::f** 只是拥有同样名字的另一个函数（重载）而不是重写。当你通过 `A` 类型的指针调用 `f()` 可能会期望打印出 `B::f`，但实际上则会打出 `f(short)` 而不是 `f(int)`。另一个很微妙的错误情况：参数相同，但是基类的函数是 `const` 的，派生类的函数却不是。

```
class A
{
public:
    virtual void f(int) const {std::cout << "A::f" << std::endl;}
};
class B : public A
{
public:
    virtual void f(int) {std::cout << "B::f" << std::endl;}
};
```

同样，这两个函数是重载而不是重写。幸运的是，现在有一种方式能描述你的意图。新标准加入了两个新的标识符（不是关键字）

**override**[派生类中使用]，表示函数必须重写基类中的虚函数，如果派生类没有重写到将编译报错。

**final**[基类中使用]，表示派生类不应当重写这个虚函数，如果派生类重写了基类的虚函数将编译报错。【在类名后使用，显示的禁止类被继承，即不能再有派生类。在虚函数后使用 `final`，显示的禁止该函数在子类里再被重写】

```
class A
{
public:
    virtual void f(short) { std::cout << "A::f" << std::endl; }
    virtual void g(int) final { std::cout << "A::g" << std::endl; }
};
class B : public A
{
public:
    virtual void f(short) override { std::cout << "B::f" << std::endl; }
    //virtual void g(int) { std::cout << "A::g" << std::endl; } // error C3248: "main
        ::A::g": final'func can not be "main::B::g" Rewrite: Wu fa Bei ChongXie
    virtual void g(float) { std::cout << "A::g" << std::endl; } // ChongZai
};
```

## 1.2.2 成员初始化

允许类在声明时使用赋值或者花括号的方式直接初始化。无须在函数里特别指定。

### 1.2.3 Defaulted

#### -引入

C++ 的类有四类特殊成员函数，它们分别是：默认构造函数、析构函数、拷贝构造函数以及拷贝赋值运算符。这些类的特殊成员函数负责创建、初始化、销毁，或者拷贝类的对象。如果程序员没有显式地为一个类定义某个特殊成员函数，而又需要用到该特殊成员函数时，则编译器会隐式的为这个类生成一个默认的特殊成员函数。如：

```
class X{
private:
    int a;
};

X x;
```

程序员并没有定义类 `X` 的默认构造函数，但是在创建类 `X` 的对象 `x` 的时候，又需要用到类 `X` 的默认构造函数，此时，编译器会隐式的为类 `X` 生成一个默认构造函数。该自动生成的默认构造函数没有参数，包含一个空的函数体，即 `X::X()`。虽然自动生成的默认构造函数仅有一个空函数体，但是它仍可用来成功创建类 `X` 的对象 `x`

但是，如果程序员为类 `X` 显式的自定义了非默认构造函数，却没有定义默认构造函数的时候，将会出现编译错误，如：

```
class X{
public:
    X(int i){
        a = i;
    }
private:
    int a;
};

X x; // 错误，默认构造函数 X::X() 不存在
```

编译出错的原因在于类 `X` 已经有了用户自定义的构造函数，所以编译器将不再会为其隐式的生成默认构造函数。如果需要用到默认构造函数来创建类的对象时，程序员必须自己显式的定义默认构造函数。例如：

```
class X{
public:
    X(){}; // 手动定义默认构造函数
    X(int i){
        a = i;
    }
private:
    int a;
};
```

```
X x; // 正确，默认构造函数 X::X() 存在
```

原本期望编译器自动生成的默认构造函数需要程序员手动编写了，即程序员的工作量加大了。此外，手动编写的默认构造函数的代码执行效率比编译器自动生成的默认构造函数低。

为了解决上述代码所示的两个问题：1. 减轻程序员的编程工作量；2. 获得编译器自动生成的默认特殊成员函数的高的代码执行效率，C++11 标准引入了一个新特性：defaulted 函数。程序员只需在函数声明后加上 “=default;”，就可将该函数声明为 defaulted 函数，编译器将为显式声明的 defaulted 函数自动生成函数体。

```
class X{
public:
    X()= default;
    X(int i){
        a = i;
    }
private:
    int a;
};

X x;
```

编译器会自动生成默认构造函数 X::X()，该函数可以比用户自己定义的默认构造函数获得更高的代码效率

## -使用

Defaulted 函数特性仅适用于类的特殊成员函数，且该特殊成员函数没有默认参数。例如：

```
class X {
public:
    int f() = default; // 错误，函数 f() 非类 X 的特殊成员函数
    X(int) = default; // 错误，构造函数 X(int, int) 非 X 的特殊成员函数
    X(int = 1) = default; // 错误，默认构造函数 X(int=1) 含有默认参数
};
```

Defaulted 函数既可以在类体里（inline）定义，也可以在类体外（out-of-line）定义。例如：

```
class X{
public:
    X() = default; //Inline defaulted 默认构造函数
    X(const X&);
    X& operator = (const X&);
    ~X() = default; //Inline defaulted 析构函数
};

X::X(const X&) = default; //Out-of-line defaulted 拷贝构造函数
X& X::operator = (const X&) = default; //Out-of-line defaulted 拷贝赋值操作符
```

在 C++ 代码编译过程中，如果程序员没有为类 X 定义析构函数，但是在销毁类 X 对象的时候又需要调用类 X 的析构函数时，编译器会自动隐式的为该生成一个析构函数。该自动生成的析构函数没有参数，包含一个空的函数体，即 `X::~X()`。例如：

```
class X {
private:
    int x;
};

class Y: public X {
private:
    int y;
};

int main()
{
    X* x = new Y;
    delete x;
}
```

在上述代码中，程序员没有为基类 X 和派生类 Y 定义析构函数，当在主函数内 `delete` 基类指针 x 的时候，需要调用基类的析构函数。于是，编译器会隐式自动的为类 X 生成一个析构函数，从而可以成功的销毁 x 指向的派生类对象中的基类子对象（即 int 型成员变量 x）。

但是，这段代码存在内存泄露的问题，当利用 `delete` 语句删除指向派生类对象的指针 x 时，系统调用的是基类的析构函数，而非派生类 Y 类的析构函数，因此，编译器无法析构派生类的 int 型成员变量 y

因此，一般情况下我们需要将基类的析构函数定义为虚函数，当利用 `delete` 语句删除指向派生类对象的基类指针时，系统会调用相应的派生类的析构函数（实现多态性），从而避免内存泄露。但是编译器隐式自动生成的析构函数都是非虚函数，这就需要由程序员手动的为基类 X 定义虚析构函数，例如：

```
class X {
public:
    virtual ~X(){}; // 手动定义虚析构函数
    //virtual ~X()= defaulted; // 或编译器自动生成 defaulted 函数定义体
private:
    int x;
};

class Y: public X {
private:
    int y;
};

int main()
{
    X* x = new Y;
```

```
        delete x;
    }
```

在上述代码中，由于程序员手动为基类 X 定义了虚析构函数，当利用 delete 语句删除指向派生类对象的基类指针 x 时，系统会调用相应的派生类 Y 的析构函数（由编译器隐式自动生成）以及基类 X 的析构函数，从而将派生类对象完整的销毁，可以避免内存泄露。

## 1.2.4 Deleted

### -引入

对于 C++ 的类，如果程序员没有为其定义特殊成员函数，那么在需要用到某个特殊成员函数的时候，编译器会隐式的自动生成一个默认的特殊成员函数，比如拷贝构造函数，或者拷贝赋值操作符。例如：

```
class X{
public:
    X();
};

int main()
{
    X x1;
    X x2=x1; // 正确，调用编译器隐式生成的默认拷贝构造函数
    X x3;
    x3=x1; // 正确，调用编译器隐式生成的默认拷贝赋值操作符
}
```

在上述代码中，程序员不需要自己手动编写拷贝构造函数以及拷贝赋值操作符，依靠编译器自动生成的默认拷贝构造函数以及拷贝赋值操作符就可以实现类对象的拷贝和赋值。这在某些情况下是非常方便省事的，但是在某些情况下，假设我们不允许发生类对象之间的拷贝和赋值，可是又无法阻止编译器隐式自动生成默认的拷贝构造函数以及拷贝赋值操作符，那这就成为一个问题了

为了能够让程序员显式的禁用某个函数，C++11 标准引入了一个新特性：deleted 函数。程序员只需在函数声明后加上 “=delete;”，就可将该函数禁用。例如，我们可以将类 X 的拷贝构造函数以及拷贝赋值操作符声明为 deleted 函数，就可以禁止类 X 对象之间的拷贝和赋值

```
class X{
public:
    X();
    X(const X&) = delete; //声明拷贝构造函数为 deleted 函数
    X& operator = (const X &) = delete; //声明拷贝赋值操作符为 deleted 函数
};

int main()
{
    X x1;
```

```

    X x2=x1; // 错误, 拷贝构造函数被禁用
    X x3;
    x3=x1; // 错误, 拷贝赋值操作符被禁用
}

```

在上述代码中, 虽然只显式的禁用了一个拷贝构造函数和一个拷贝赋值操作符, 但是由于编译器检测到类 X 存在用户自定义的拷贝构造函数和拷贝赋值操作符的声明, 所以不会再隐式的生成其它参数类型的拷贝构造函数或拷贝赋值操作符, 也就相当于类 X 没有任何拷贝构造函数和拷贝赋值操作符, 所以对象间的拷贝和赋值被完全禁止了

## -使用

Deleted 函数特性还可用于禁用类的某些转换构造函数, 从而避免不期望的类型转换。在下列代码中, 假设类 X 只支持参数为双精度浮点数 double 类型的转换构造函数, 而不支持参数为整数 int 类型的转换构造函数, 则可以将参数为 int 类型的转换构造函数声明为 deleted 函数:

```

class X{
public:
    X(double);
    X(int) = delete;
};

int main()
{
    X x1(1.2);
    X x2(2); // 错误, 参数为整数 int 类型的转换构造函数被禁用
}

```

Deleted 函数特性还可以用来禁用某些用户自定义的类的 new 操作符, 从而避免在自由存储区创建类的对象。例如:

```

#include <cstddef>
using namespace std;

class X{
public:
    void *operator new(size_t) = delete;
    void *operator new[](size_t) = delete;
};

int main()
{
    X *pa = new X; //错误, new 操作符被禁用
    X *pb = new X[10]; //错误, new[] 操作符被禁用
}

```

必须在函数第一次声明的时候将其声明为 deleted 函数, 否则编译器会报错。即对于类的成员函数而言, deleted 函数必须在类体里 (inline) 定义, 而不能在类体外 (out-of-line) 定



义。例如：

```
class X {
public:
    X(const X&);
};

X::X(const X&) = delete; //错误，deleted 函数必须在函数第一次声明处声明
```

虽然 defaulted 函数特性规定了只有类的特殊成员函数才能被声明为 defaulted 函数，但是 deleted 函数特性并没有此限制。非类的成员函数，即普通函数也可以被声明为 deleted 函数。例如：

```
int add (int,int)=delete;

int main()
{
    int a, b;
    add(a,b); // 错误，函数 add(int, int) 被禁用
}
```

### 1.2.5 委托构造

有的时候我们会声明多个不同形式的构造函数，用于在不同情况下创建对象。这些代码大都是初始化成员变量，非常类似，仅有少量的不同，但代码却并不能复用，导致代码冗余。

常用的解决方法是实现一个特殊的初始化函数（通常名字为 init），然后在每个构造函数里调用它。如：

```
class Demo{
private:
    int x,y;
    void init(int a,int b){x = a; y = b;}

public:
    Demo(){init(0,0);}
    Demo(int a){init(a,0);}
    Demo(int a, int b){init(a,b);}
};
```

C++ 引入委托构造概念，解决方法基本相同，但不必再专门写一个特殊的初始化函数，而是可以直接调用本类的其他构造函数，把对象的构造工作委托给其他的构造函数完成。

```
class Demo{
public:
    Demo():Demo(0,0){}
    Demo(int a):Demo(a,0){}
    Demo(int a, int b){x = a; y = b;}
};
```

## 1.3 类型改进

### 1.3.1 类型推导

-auto

自动推测变量的类型

```
for(auto it=vec.begin();it!=vec.end();++it)
/* as the following code */
//for(std::vector<int>::iterator it=vec.begin();it!=vec.end();++it)

/* of Course, we can reference too*/
int arr[ ] = {1,2,3,4,5};
for(auto& e : arr)
{
    e = e*e;
}
```

extends: 根据 auto 变量的类型声明一个相同类型的数据变量。

```
double db = 10.9;
double *pdb = &db;
auto num = pdb;//in interface

// typeid().name use to tell the value's type
std::cout << typeid(db).name() << std::endl;
std::cout << typeid(num).name() << std::endl;
std::cout << typeid(pdb).name() << std::endl;

//typeid(db).name() db2;
//According the db value to define a same type value db2;
decltype(db) numA(10.9);//stock interface
std::cout << sizeof(numA) <<"\000"<< numA << std::endl;
```

-decltype

### 1.3.2 类型别名

- 可以完成与 typedef 相同的工作, 使用 “using alias = type;” 的形式为类型起别名。
- 可以结合 template 关键字为模版类声明 “部分特化” 的别名

```
template <typename T>
using int_map = std::map<int,T>; //固定key值为int

int_map<string> m; //使用别名, 省略了一个参数, typedef 同可以做到

template <typename First, typename Second, int Third>
```

```

class SomeType;

template <typename Second>
typedef SomeType<OtherType, Second, 5> TypedefName; // Illegal in C++03

template <typename Second>
using TypedefName = SomeType<OtherType, Second, 5>; // Legal in C++11

```

- 函数指针方式，更明了

```

typedef void (*FunctionType)(double); // Old style
using FunctionType = void (*)(double); // New introduced syntax

```

### 1.3.3 nullptr

以前都是用 0 来表示空指针的，但由于 0 可以被隐式类型转换为整形，这就会存在一些问题。关键字 nullptr 是 std::nullptr\_t 类型的值，用来指代空指针。nullptr 和任何指针类型以及类成员指针类型的空值之间可以发生隐式类型转换，同样也可以隐式转换为 bool 型（取值为 false）。但是不存在到整形的隐式类型转换

```

void foo(int* p) {}
void bar(std::shared_ptr<int> p) {}
int* p1 = NULL;
int* p2 = nullptr;
if(p1 == p2) {}
foo(nullptr);
bar(nullptr);
bool f = nullptr;
int i = nullptr; // error: A native nullptr can only be converted to bool or, using
                 reinterpret_cast, to an integral type

```

### 1.3.4 constexpr

相当于编译期的常量，让所修饰的表达式或函数具有编译的常量性，可以让编译器更好的优化代码。

### 1.3.5 R-raw string

A raw string literal starts with R”( and ends in )”

```

int main()
{
    string normal_str="First_line.\nSecond_line.\nEnd_of_message.\n";
    string raw_str=R"(First_line.\nSecond_line.\nEnd_of_message.\n)";
    cout<<normal_str<<endl;
    cout<<raw_str<<endl;
}

```

```

    return(0);
}

/*
    output

    -->
    First line.
    Second line.
    End of message.

    First line.\nSecond line.\nEnd of message.\n
*/

```

### 1.3.6 Uniform Initialization

统一使用{...} 初始化

```

CircleStruct myCircle5{10, 10, 2.5};
CircleClass myCircle6{10, 10, 2.5};

CircleStruct myCircle3 = {10, 10, 2.5};
CircleClass myCircle4 = {10, 10, 2.5};

int a = 3;
int b(3);
int c = {3}; // Uniform initialization
int d{3}; // Uniform initialization

int e{}; // Uniform initialization, e will be 0

```

### 1.3.7 Initialize List

```

#include <initializer_list>
using namespace std;
int makeSum(initializer_list<int> lst)
{
    int total = 0;
    for (const auto& value : lst) {
        total += value;
    }
    return total;
}

int a = makeSum({1,2,3});
int b = makeSum({10,20,30,40,50,60});

```

### 1.3.8 Explicit Conversion Operators

```
class IntWrapper
{
public:
    IntWrapper(int i) : mInt(i) {}
    operator int() const { return mInt; }
private:
    int mInt;
};

// The following code demonstrates this implicit conversion; iC1 will contain the
// value 123:
IntWrapper c(123);
int iC1 = c;
int iC1 = static_cast<int>(c); // Ok

class IntWrapper
{
public:
    IntWrapper(int i) : mInt(i) {}
    explicit operator int() const { return mInt; }
private:
    int mInt;
};

IntWrapper c(123);
int iC1 = c; // Error, because of explicit int() operator
int iC2 = static_cast<int>(c); // Ok
```

### 1.3.9 Smart Pointers

现在能使用的，带引用计数，并且能自动释放内存的智能指针包括以下几种：

#### shared\_ptr

shared\_ptr 采用引用计数的方式管理所指向的对象。当有一个新的shared\_ptr 指向同一个对象时（复制shared\_ptr 等- 传参也是一种如func(p),p 的引用计数 +1），引用计数加 1。当shared\_ptr 离开作用域时，引用计数减 1。当引用计数为 0 时，释放所管理的内存。

这样做的好处在于解放了程序员手动释放内存的压力。之前，为了处理程序中的异常情况，往往需要将指针手动封装到类中，通过析构函数来释放动态分配的内存；现在这一过程就可以交给shared\_ptr 去做了。

引用计数实现 中间对象:<http://blog.csdn.net/jiangfuqiang/article/details/8292906>  
<http://www.cnblogs.com/helloamigo/p/3575098.html>

一般我们使用 `make_shared` 来获得 `shared_ptr`

```
shared_ptr<string> p1 = make_shared<string>("");
if(p1 && p1->empty())
    *p1 = "hello";

auto p2 = make_shared<string>("world");
cout<<*p1<<'_'<<*p2<<endl;

cout<<"test_shared_ptr_use_count:"<<endl;
cout<<"p1_cnt:"<<p1.use_count()<<"\tp2_cnt:"<<p2.use_count()<<endl;

auto p3 = p2;
cout<<"p1_cnt:"<<p1.use_count()<<"\tp2_cnt:"<<p2.use_count()<<"\tp3_cnt:"<<p3.
    use_count()<<endl;
p2 = p1;
cout<<"p1_cnt:"<<p1.use_count()<<"\tp2_cnt:"<<p2.use_count()<<"\tp3_cnt:"<<p3.
    use_count()<<endl;
```

可以使用一个 `new` 表达式返回的指针进行初始化

```
shared_ptr<int> p4(new int(1024));
//shared_ptr<int> p5 = new int(1024); // wrong, no implicit constructor
cout<<*p4<<endl;

int *p7 = new int(1024);
shared_ptr<int> p8(p7);
```

可以通过 `reset` 方法重置指向另一个对象, 此时原对象的引用计数减一

```
p1.reset(new string("cpp11"));
```

可以定制一个 `deleter` 函数, 用于在 `shared_ptr` 释放对象时调用

```
void print_at_delete(int *p)
{
    cout<<"deleting..."<<p<<'\\t'<<*p<<endl;
    delete p;
}

cout<<"test_shared_ptr_deleter:"<<endl;
int *p7 = new int(1024);
shared_ptr<int> p8(p7, print_at_delete);
p8 = make_shared<int>(1025);
```

## unique\_ptr

`unique_ptr` 对于所指向的对象, 正如其名字所示, 是独占的。所以, 不可以对 `unique_ptr` 进行拷贝、赋值等操作, 但是可以通过 `release` 函数在 `unique_ptr` 之间转移控制权。

```

unique_ptr<int> up1(new int(1024));
cout<<"up1:␣"<<*up1<<endl;
unique_ptr<int> up2(up1.release());
cout<<"up2:␣"<<*up2<<endl;
//unique_ptr<int> up3(up1); // wrong, unique_ptr can not copy
//up2 = up1; // wrong, unique_ptr can not copy
unique_ptr<int> up4(new int(1025));
up4.reset(up2.release());
cout<<"up4:␣"<<*up4<<endl;

```

**作为参数和返回值的唯一方式** 上述对于拷贝的限制，有两个特殊情况，即unique\_ptr 可以作为函数的返回值和参数使用，这时虽然也有隐含的拷贝存在，但是并非不可行的。

```

unique_ptr<int> clone(int p)
{
    return unique_ptr<int>(new int(p));
}

void process_unique_ptr(unique_ptr<int> up)
{
    cout<<"process_unique_ptr:␣"<<*up<<endl;
}

auto up5 = clone(1024);
cout<<"up5:␣"<<*up5<<endl;

process_unique_ptr(std::move(up5));

```

## weak\_ptr

weak\_ptr 一般和shared\_ptr 配合使用。它可以指向shared\_ptr 所指向的对象，但是却不增加对象的引用计数。这样就有可能出现weak\_ptr 所指向的对象实际上已经被释放了的情况。因此，weak\_ptr 有一个lock 函数，尝试取回一个指向对象的shared\_ptr。

```

auto p10 = make_shared<int>(1024);
weak_ptr<int> wp1(p10);
cout<<"p10␣use_count:␣"<<p10.use_count()<<endl;

//p10.reset(new int(1025)); // this will cause wp1.lock() return a false obj
shared_ptr<int> p11 = wp1.lock();
if(p11) cout<<"wp1:␣"<<*p11<<"␣use_count:␣"<<p11.use_count()<<endl;

```

## 1.4 处理方式 (函数) 改进

### 1.4.1 匿名函数 (也叫 Lambda 表达式)

`[capture](parameters)->return-type{body}`

<http://www.cnblogs.com/haippy/archive/2013/05/31/3111560.html>

capture 指定了在可见域范围内 lambda 表达式的代码内可见得外部变量的列表

类型 1: `[](parameters){body}`

- `[](int x, int y){ return x + y;}` // 隐式返回类型
- `[](int& x){ ++x;}` // 没有 return 语句 -> lambda 函数的返回类型是 'void'
- `[](){ ++global_x;}` // 没有参数, 仅访问某个全局变量
- `[]{ ++global_x;}` // 与上一个相同, 省略了 ()

类型 2: `[](int x, int y) -> int { int z = x + y; return z; }` 在这个例子中创建了一个临时变量 `z` 来存储中间值. 和普通函数一样, 这个中间值不会保存到下次调用. 什么也不返回的 Lambda 函数可以省略返回类型, 而不需要使用 `-> void` 形式.

类型 3: 闭包操作 Lambda 函数可以引用在它之外声明的变量. 这些变量的集合叫做一个闭包. 闭包被定义在 Lambda 表达式声明中的方括号 `[]` 内. 这个机制允许这些变量被按值或按引用捕获

- `[]` //未定义变量. 试图在 Lambda 内使用任何外部变量都是错误的.
- `[x,&y]` //x 按值捕获, y 按引用捕获.
- `[&]` //用到的任何外部变量都隐式按引用捕获
- `[=]` //用到的任何外部变量都隐式按值捕获
- `[&,x]` //x 显式地按值捕获. 其它变量按引用捕获
- `[=,&z]` //z 按引用捕获. 其它变量按值捕获

example1:

```
//[capture list] (parameter list) -> return type { function body }

std::vector<int> some_list;
int total = 0;
for (int i=0;i<5;++i) some_list.push_back(i);
std::for_each(begin(some_list), end(some_list), [&total](int x)
{
    total += x;
}); //func pointer
```



example2:

```
std::vector<int> some_list;
int total = 0;
int value = 5;
std::for_each(begin(some_list), end(some_list), [&, value, this](int x)
{
    total += x * value * this->some_func();
});
```

## 1.4.2 for 的变革

增加for..each

```
void funcX()
{
    std::vector<int> vec;
    ...
    for(int elem:vec)
    {
        ...
    }
    /* as follow */
    /* for(std::vector<int>::iterator it=vec.begin();it!=vec.end();++it)
    * {
    *   int elem=*it;
    *   ...
    * }
    */
}
```

## 1.4.3 非成员 begin() 和 end()

他们是新加入标准库的，除了能提高了代码一致性，还有助于更多地使用泛型编程。它们和所有的 STL 容器兼容。更重要的是，他们是可重载的。所以它们可以被扩展到支持任何类型。对 C 类型数组的重载已经包含在标准库中了。

```
int arr[] = {1,2,3};
std::for_each(&arr[0], &arr[0]+sizeof(arr)/sizeof(arr[0]), [](int n) {std::cout << n
    << std::endl;});
auto is_odd = [](int n) {return n%2==1;};
auto begin = &arr[0];
auto end = &arr[0]+sizeof(arr)/sizeof(arr[0]);
auto pos = std::find_if(begin, end, is_odd);
if(pos != end)
    std::cout << *pos << std::endl;
```

使用非成员的begin() 和 end():

```
int arr[] = {1,2,3};
std::for_each(std::begin(arr), std::end(arr), [](int n) {std::cout << n << std::endl;});
auto is_odd = [](int n) {return n%2==1;};
auto pos = std::find_if(std::begin(arr), std::end(arr), is_odd);
if(pos != std::end(arr))
    std::cout << *pos << std::endl;
```

这基本上和使用std::vector 的代码是完全一样的。这就意味着我们可以写一个泛型函数处理所有支持begin() 和end() 的类型。

```
template <typename Iterator>
void bar(Iterator begin, Iterator end)
{
    std::for_each(begin, end, [](int n) {std::cout << n << std::endl;});
    auto is_odd = [](int n) {return n%2==1;};
    auto pos = std::find_if(begin, end, is_odd);
    if(pos != end)
        std::cout << *pos << std::endl;
}

template <typename C>
void foo(C c)
{
    bar(std::begin(c), std::end(c));
}

template <typename T, size_t N>
void foo(T(&arr)[N])
{
    bar(std::begin(arr), std::end(arr));
}

int arr[] = {1,2,3};
foo(arr);
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
foo(v);
```

#### 1.4.4 静态断言

C 语言提供断言assert，它是一个宏，可以在运行时验证某些条件是否成立，有利于保证程序的正确性。

但是，泛型主要工作在编译器，assert 无法起作用。

static\_assert 就是解决这个问题的，他是编译器的断言，可以在编译器加入诊断信息，提前检查可能发生的错误。

`static_assert`: 需要一个编译期的bool 表达式和警告消息字符串 -

```
static_assert(condition,message);
```

如果bool 表达式在编译期的计算结果为false, 那么编译器会报错, 并给出message.-

```
static_assert(sizeof(int) == 4, "int must be 32bit!");
```

一般配合 `type_traits` 库使用。

## 1.5 参考

[https://en.wikipedia.org/wiki/C%2B%2B11#Template\\_aliases](https://en.wikipedia.org/wiki/C%2B%2B11#Template_aliases)



## 第二章 return 语句

### 2.1 传值返回

将 i 先拷贝的临时存储空间，调用着获得的是 i 的一个副本

### 2.2 引用返回

将 i 的值直接拷贝到接受的空间中（调用者）

如果一个函数以引用返回，则这个函数调用可出现在赋值语句的左边

因为使用引用返回的函数返回的是一个实际单元，{必须保证函数返回时该单元仍然有效（不能是临时变量）}

例子

错误：引用返回了一个临时变量，临时变量返回值是一个值的拷贝，没有实际空间

```
int& functionForTestReference()
{
    int i;
    return i;
}
```

正确：对于临时变量的值是可以复制过来，但是没有实际空间，或内存地址

```
int functionForTestLocalVariable()
{
    int i;
    return i;
}
```



## 第三章 inline 关键字

### 3.1 使用

在函数定义中函数返回类型前加上关键字 `inline`, 注: 声明不起作用

```
inline int min(int first,int secend){...}
```

关键字`inline` 必须与函数定义体放在一起才能使函数成为内联, 仅将`inline` 放在函数声明前面不起任何作用。

即 `inline` 只有与实现放在一起时 才能真正的内联

定义在类声明之中的成员函数将自动地成为内联函数

例如

```
class A
{
    public: void Foo(int x, int y) { } // 自动地成为内联函数
}
```

将成员函数的定义体放在类声明之中虽然能带来书写上的方便, 但不是一种良好的编程风格, 上例应该改成:

// 头文件

```
class A
{
public:
    void Foo(int x, int y);
}

// 定义文件
inline void A::Foo(int x, int y){ ...在头文件中实现...}
```

### 3.2 效率

#### 3.2.1 实现机制

内联是以代码膨胀(复制)为代价, 仅仅省去了函数调用的开销, 从而提高函数的执行效率。如果执行函数体内代码的时间, 相比于函数调用的开销较大, 那么效率的收获会很少。另一方面,

每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

### 3.2.2 慎用内联

- 如果函数体内的代码比较长，使用内联将导致内存消耗代价较高
- 如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如“偷偷地”执行了基类或成员对象的构造函数和析构函数。所以不要随便地将构造函数和析构函数的定义体放在类声明中。一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这进一步说明了 `inline` 不应该出现在函数的声明中）。

- 如果函数体内存在递归
- 如果函数体内存在数组
- 如果函数体内存在 `goto`, `switch` 语句

## 3.3 内联技巧

### 3.3.1 条件内联

如果想用一个预编译选项来控制某些函数何时内联，何时关闭内联，可以使用条件内联的技巧。

将内联函数的定义放到 `.inl` 中，其它函数的定义放到 `.cpp` 中，然后在 `.h` 中加入：

```
// inline.h
#ifdef INLINE
    #include "*.inl"
#endif
```

在 `.inl` 中加入：

```
#ifndef INLINE
#define inline // let inline be void
#endif
inline FuncX(...) {}
```

在 `.cpp` 中加入：

```
#ifndef INLINE
    #include "inline.h"
#endif
```

### 3.3.2 选择性内联

可以将某函数在一些调用点处内联，而在其它调用点处不内联



## 3.4 结论

内联函数并不是一个增强性能的灵丹妙药。只有当函数非常短小的时候它才能得到我们想要的效果，但是如果函数并不是很短而且在很多地方都被调用的话，那么将会使得可执行体的体积增大。最令人烦恼的还是当编译器拒绝内联的时候。在老的实现中，结果很不尽人意，虽然在新的实现中有很大的改善，但是仍然还是不那么完善的。一些编译器能够足够的聪明来指出哪些函数可以内联哪些不能，但是，大多数编译器就不那么聪明了，因此这就需要我们的经验来判断。如果内联函数不能增强性能，就避免使用它！



## 第四章 重载

<http://blog.csdn.net/zx3517288/article/details/48976097>

### 4.1 函数重载

依赖于参数，与返回值无关

1. 在一个类中定义了多个同名的方法
2. 函数名相同
3. 与参数个数与类型有关
4. 与返回值无关

### 4.2 函数重载后的隐藏规则

1. 是指派生类的函数屏蔽了 与其同名的基类函数，注意只要同名函数，不管参数列表是否相同，基类函数都会被隐藏。

```
#include <iostream>
using namespace std;

class Base
{
public:
    void fun(double ,int ){ cout << "Base::fun(double_,int_)" << endl; }
};

class Derive : public Base
{
public:
    void fun(int ){ cout << "Derive::fun(int_)" << endl; }
};

int main()
{
```

```

    Derive pd;
    pd.fun(1);//Derive::fun(int )
    pb.fun(0.01, 1);//error C2660: "Derive::fun" : 函数不接受 2 个参数

    return 0;
}

```

## 4.3 运算符重载

### 4.3.1 重载 = 运算符

防止浅拷贝造成的析构错误.

1. `Class-name & operator=(const Class-name & RightValue)`
2. rewrite the Func

### 4.3.2 重载 'output' 运算符

1. `friend ostream & operator <<(ostream & outputStream ,const Class-name & RightValue)`
2. rewrite the Func

### 4.3.3 重载 New Delete 运算符

1. `static void *operator new(size_t size)`
2. rewrite the Func `+= ::new` 构造函数. (全局new 劫持, 进行对象内存分配等操作)

### 4.3.4 重载 ++ 运算符

1. 类名 `&operator++();` //匹配 `++i`
2. 类名 `operator++(int);` //匹配 `i++`

## 4.4 函数的默认参数

1. 默认参数必须放在右边
2. 默认参数中间不允许出现不默认的
3. 函数指针没有默认参数, 必须全部输入数据
4. 函数重载与函数默认参数冲突, 需要你输入的参数类型不一个, 个数不一样, 顺序不一样不会出现问题, 否则一定报错

```
void print(int c,int a = 1, int d=2, int b = 3)
{
    std::cout << a<<b<<c << std::endl;
}

void(*pt1)(int c, int a , int d , int b ) = print;
pt1(100,1,2,3);//Func Pointer
```



## 第五章 struct And union

### 5.1 Struct

#### 5.1.1 定义

struct 又名联合体，假设定义结构为

```
struct student
{
    char mark;
    long num;
    float score;
};
```

则其结构如下：

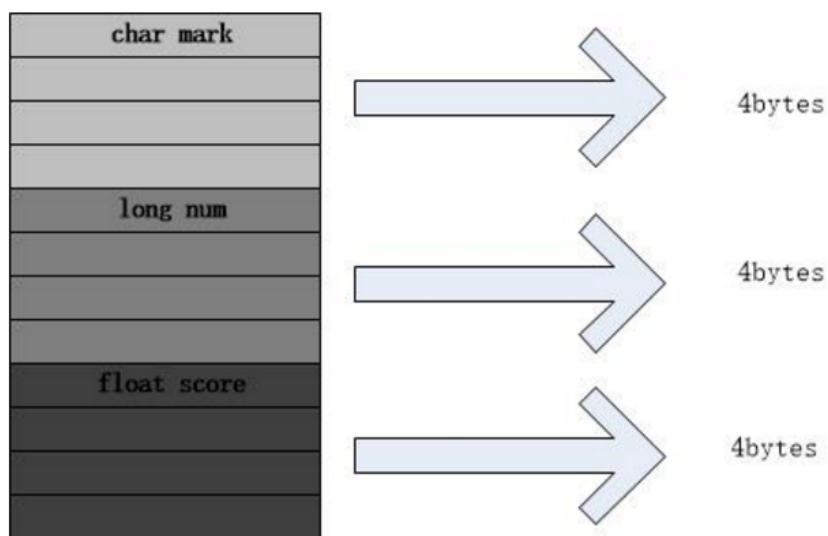


图 5.1: Struct 结构图

sizeof(struct student) 的值为 12bytes.

#### 5.1.2 函数使用

传值

1. 使用引用改变 void Move(struct & Parameter)

## 2. 使用指针改变 void Move(struct \* Parameter)

### 初始化

```
struct ListNode{
    int val;
    ListNode* next; // 不用加struct 关键字
    ListNode(int i):val(i),next(nullptr){} // 可以与类一样的定义构造函数
};

ListNode * ptr = new ListNode(0); //不用typedef 就可以当类型用，如同类，可以理解为公开的
类
```

### 5.1.3 变长结构体

```
struct MyData
{
    int nLen;
    char data[0];
};

sizeof(MyData)=4;
```

sizeof 等于 4 很纳闷吧，可能有的编译器不支持char data[0];需要用char data[1];代替，这样上面结构体大小是sizeof(MyData)=8(字节对齐)；

在上结构中,data是一个数组名;但该数组没有元素;该数组的真实地址紧随结构体 MyData 之后，而这个地址就是结构体后面数据的地址（如果给这个结构体分配的内容大于这个结构体实际大小，后面多余的部分就是这个 data 的内容），这种声明方法可以巧妙的实现 C 语言里的数组扩展。

```
struct MyData
{
    int nLen;
    char data[0];
};

typedef struct
{
    int data_len;
    char *data;
}buff_st_2;

char str[10] = "123456789";
cout << "Size_of_MyData:" << sizeof(MyData) << endl;
cout << "Size_of_buff_st_2:" << sizeof(buff_st_2) << endl;
MyData *myData = (MyData*)malloc(sizeof(MyData) + 10);
```



```
memcpy(myData->data, str, 10);
cout << "myData's Data is:" << myData->data << endl;

//Size of MyData: 4
//Size of buff_st_2: 8
//myData's Data is: 123456789
```

注意 data[0]和data[] 不占用空间，且地址紧跟在结构后面，而char \*data 作为指针，占用4个字节，地址不在结构之后

<https://blog.csdn.net/nbaDWde/article/details/79722403>

## 5.2 Union

### 5.2.1 定义

union 又名共用体，假设定义结构为

```
union test
{
    char mark;
    long num;
    float score;
};

union DATE
{
    char a;
    int i[5];
    double b;
};
```

则其结构如下：

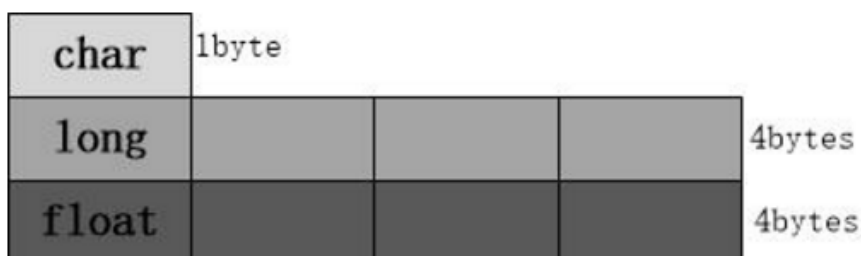


图 5.2: Union test 结构图

sizeof(union test) 的值为 4bytes. 原因如下: 有的时候，我们需要几种不同类型的变量存在在同一段的内存空间中，就像上面的，我们需要将一个 char 类型的 mark、一个 long 类型的 num 变量和一个 float 类型的 score 变量存放在同一个地址开始的内存单元中。

上面的三个变量，char 类型和 long 类型所占的内存字节数是不一样的，但是在 union 中，它们都是从同一个地址存放的，也就是使用的覆盖技术，这三个变量互相覆盖，而这种使几个不同的变量共占同一段内存的结构，称为“共用体”类型的结构

结构体 struct 所占用的内存为各个成员的占用的内存之和（当然也需要考虑内存对齐的问题了）。而对于 union 来说，在谭浩强的《C 语言程序设计》中这么说：**union 变量所占用的内存长度等于最长的成员的内存长度。**

它的所有成员相对于基地址的偏移量都为 0。

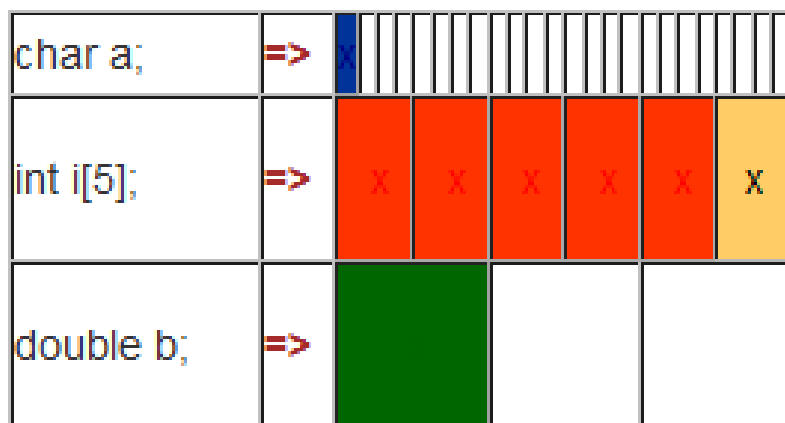


图 5.3: Union Date 结构图

该结构要放得下 int i[5] 必须要至少占  $4 \times 5 = 20$  个字节。如果没有 double 的话 20 个字节够用了，此时按 4 字节对齐。但是加入了 double 就必须考虑 double 的对齐方式，double 是按照 8 字节对齐的，所以必须添加 4 个字节使其满足  $8 \times 3 = 24$ ，也就是必须也是 8 的倍数，这样一来就出来了 24 这个数字。综上所述，最终联合体的最小的 size 也要是所包含的所有类型的基本长度的最小公倍数才行。（这里的字节数均指 win 下的值，平台、编译器不同值也有可能不同。）

## 5.3 字节对齐-StructAndUnion

不同编译器环境默认字节对齐的差别，做平台移植的同仁要注意了，遇到不确定的字节对齐问题，最好先亲自试一下，不能太想当然了：

1. Win32 下，VC 编译器默认 8 字节对齐，而且支持 1、2、4、8、16 五种对齐方式
2. Linux 32 下，GCC 4.1 默认 4 字节对齐，支持 1、2、4 三种对齐方式。因此结构体中即使遇到 double、long long 这样的 8 字节变量，仍然按 4 字节对齐。即使设定了 #pragma pack(8)，但在 win32 下是 8 字节

## 第六章 string

### 6.1 方法

#### 6.1.1 string 类的构造函数

```
#include <iostream>
#include <cassert>
#include <iterator>
#include <string>

int main()
{
    // 默认缺省构造
    {
        // string::string()
        std::string s;
        assert(s.empty() && (s.length() == 0) && (s.size() == 0));
    }

    // n 个相同字符
    {
        // string::string(size_type count, charT ch)
        std::string s(4, '=');
        std::cout << s << '\n'; // "===="
    }

    {
        std::string const other("Exemplary");
        // string::string(string const& other, size_type pos, size_type count)
        std::string s(other, 0, other.length()-1);
        std::cout << s << '\n'; // "Exemplar"
    }

    {
        // string::string(charT const* s, size_type count)
        std::string s("C-style_string", 7);
        std::cout << s << '\n'; // "C-style"
    }
}
```

```

{
    // string::string(charT const* s)
    std::string s("C-style\0string");
    std::cout << s << '\n'; // "C-style"
}

{
    char mutable_c_str[] = "another_C-style_string";
    // string::string(InputIt first, InputIt last)
    // 左闭右开
    std::string s(std::begin(mutable_c_str)+8, std::end(mutable_c_str)-1);
    std::cout << s << '\n'; // "C-style string"
}

{
    std::string const other("Exemplar");
    std::string s(other);
    std::cout << s << '\n'; // "Exemplar"
}

{
    // string::string(string&& str)
    std::string s(std::string("C++by") + std::string("example"));
    std::cout << s << '\n'; // "C++ by example"
}

{
    // string(std::initializer_list<charT> ilist)
    std::string s({ 'C', '-', 's', 't', 'y', 'l', 'e' });
    std::cout << s << '\n'; // "C-style"
}
}

```

### 6.1.2 string 类的字符操作

- `operator[n]` 和 `at(n)` 均返回当前字符串中第 `n` 个字符的位置，但 `at` 函数提供范围检查，当越界时会抛出 `out_of_range` 异常，下标运算符 `[]` 不提供检查访问
- `bool empty() const`; Checks if the string has no characters, i.e. whether `begin() == end()`
- `const char *data() const`; // 返回一个非 null 终止的不可修改的 c 字符数组
- `const char *c_str() const`; // 返回一个以 null 终止的不可修改的 c 字符串
- `int copy(char *s, int n, int pos = 0) const`; // 把当前串中以 `pos` 开始的 `n` 个字符拷贝到以 `s` 为起始位置的字符数组中，返回实际拷贝的数目

- `void push_back( CharT ch );` Appends the given character `ch` to the end of the string
- `void pop_back();` Removes the last character from the string.
- `stoi()`, `stof()` 将字符串转换为整数和浮点数
- `std::to_string(num_type)` 将数字转换为字符串

## 6.2 操作符

### 6.2.1 += 操作符

左边只能是string字符串，右边可以是 string 字符串，C 风格的字符串、或是一个字符

### 6.2.2 + 操作符

+ 操作符的右边的两个不能都是字符串，至少存在一个字符串对象

```
s1 = "W.C." + "Fields"; //ERROR
```

### 6.2.3 == 操作符

```
bool operator==(const string \&s1,const string \&s2)const;//比较两个字符串是否相等
```

运算符">","<",">=","<=","!=" 均被重载用于字符串的比较;

## 6.3 宽字符 wchar\_t

1. `#include 1.stdlib.h 2.locale`
2. `setlocale(LC_ALL,"chs");`
3. `wchar_t *p = L"中文";`



# 第七章 class

## 7.1 C++ 中使用 Struct 创建 1 个类

`struct` 关键字创建的类，类成员默认是公有的，但可以有私有的  
`new` 一个对象时，只为类中成员变量分配空间，对象之间共享成员函数。

## 7.2 C++ 中使用 Class 创建 1 个类

`class` 关键字创建的类，类成员默认是私有的

## 7.3 const 型成员函数

```
int getAge() const;
```

- (1) 会隐含第一个参数为 `this`, 即如果没有参数，实际类型为带有本身类型的一个参数。
- (2) 用于限制该函数，不允许该函数对任何数据成员进行修改。
- (3) 一个 `const` 成员函数仅能调用其他 `const` 成员函数，因为 `const` 成员不允许直接或间接的改变对象的状态，而调用非 `const` 成员函数可能会间接的改变对象的状态
- (4) 默认参数应在函数声明而非函数定义中给出：全部函数

## 7.4 析构函数与构造函数

### 7.4.1 潜规则

- (1) 构造函数和析构函数比较特殊：在调用它们时不需要显式地提供函数名，编译器会自动调用它们
- (2) 构造函数不能有返回值
- (3) 构造函数主要用来对数据成员进行初始化，并负责其他一些在对象创建时需要处理的事务
- (4) 自己定义构造函数，否则编译器会生成一个公有的构造函数

## (5) 构造函数不能是虚函数

- 从存储空间角度: 虚函数对应一个 vtable, 这大家都知道, 可是这个 vtable 其实是存储在对象的内存空间的。问题出来了, 如果构造函数是虚的, 就需要通过 vtable 来调用, 可是对象还没有实例化, 也就是内存空间还没有, 无法找到 vtable, 所以构造函数不能是虚函数。

- 从使用角度: 虚函数主要用于在信息不全的情况下, 能使重载的函数得到对应的调用。构造函数本身就是初始化实例, 那使用虚函数也没有实际意义呀。所以构造函数没有必要是虚函数。

- 从实现上看: vtbl 在构造函数调用后才建立, 因而构造函数不可能成为虚函数

## 7.4.2 Initializer-List Constructors

```
class EvenSequence
{
public:
    EvenSequence(initializer_list<double> args)
    {
        if (args.size() % 2 != 0) {
            throw invalid_argument("initializer_list should "
                                   "contain even number of elements.");
        }
        mSequence.reserve(args.size());
        for (auto value : args) {
            mSequence.push_back(value);
        }
    }
    void dump() const
    {
        for (auto value : mSequence) {
            cout << value << ", ";
        }
        cout << endl;
    }
private:
    vector<double> mSequence;
};

EvenSequence p1 = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
p1.dump();
try {
    EvenSequence p2 = {1.0, 2.0, 3.0};
} catch (const invalid_argument& e) {
    cout << e.what() << endl;
}
```



### 7.4.3 初始化顺序

1. 普通的变量：一般不考虑啥效率的情况下可以在构造函数中进行赋值。考虑一下效率的可以再构造函数的初始化列表中进行
2. static 静态变量
3. const 常量变量: const 常量需要在声明的时候即初始化。一般采用在构造函数的初始化列表中进行。
4. Reference 引用型变量：引用型变量和 const 变量类似。需要在创建的时候即进行初始化。也是在初始化列表中进行。但需要注意用 Reference 类型。
5. 字符串初始化

## 7.5 值语义与对象语义

<http://blog.csdn.net/chdhust/article/details/9927083>

### 7.5.1 值语义

所谓值语义是一个对象被系统标准的复制方式复制后，与被复制的对象之间毫无关系，可以彼此独立改变互不影响

### 7.5.2 对象语义

也叫指针语义，引用语义等，通常是指一个对象被系统标准的复制方式复制后，与被复制的对象之间依然共享底层资源，对任何一个的改变都将改变另一个

## 7.6 拷贝构造函数

拷贝构造函数创建一个新的对象，此对象是另外一个对象的拷贝品

(1) `Person(Person &);` 或 `Person(const Person &);`

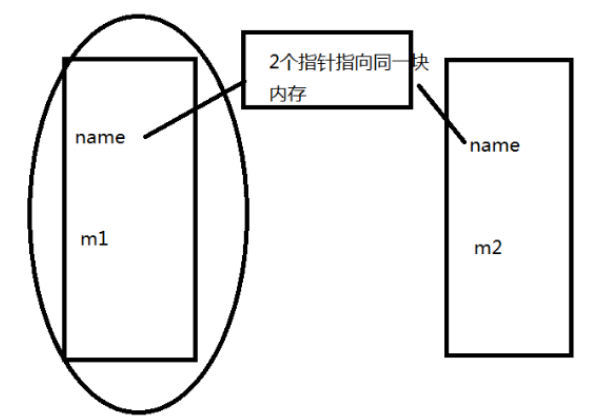
(2) 如果类的设计者不提供拷贝构造函数，编译器会自动生成一个：将源对象所有数据成员的值逐一赋值给目标对象相应的数据成员

(3) 通常，如果一个类包含指向动态存储空间指针类型的数据成员，则就应该为这个类设计拷贝构造函数（如果某个类定义了拷贝构造函数，类的设计者通常再为它添加一个赋值操作符重载函数）

(4) 浅拷贝即编译器提供的，只会拷贝值，如果是数组的话，则是该数组的引用，而不重新分配空间。

浅拷贝与深拷贝：

1- 浅拷贝：两个对象之间成员变量简单的赋值。



2- 深拷贝：不同的对象指针成员指向不同的内存地址，拷贝构造的时候不是简单的指针赋值，而是将内存拷贝过来。

**拷贝构造函数可能的错误** 带实习的时候发现，一个同学写了一个类，没有写拷贝构造函数，然后在定义函数传参数时，用到该类型本身，结果就导致析构错误。1 种错误可能是因为该类型本身用了数组 (默认拷贝构造函数数组是传引用的)，导致释放重释放错误.. 非法操作地址

2 可能是函数中 (碰到return;) 的临时变量需要调用析构函数，而参数到函数结束时也会调用析构函数。

## 7.7 初始化列表

初始化列表仅在构造函数中有效，不能用于其他函数。构造函数的初始化列表可以初始化任何数据成员，但const 类型数据成员不能用其他办法进行初始化

数据成员初始化顺序完全取决于它们在类中声明的顺序 (不是对应关系，是赋值的先后，对应还是一一对应)，与它们在初始化段中出现的次序无关

## 7.8 一些只有

- 只能在成员函数 或者 friend 函数中访问类的 非公有成员
- 类成员或者非类成员 只有 静态变量被默认初始化
- const 成员 必须在初始化段 (列表) 中被初始化

```
class Test{
public:
    // ok: 必须这样
    Test() :testConst(1){}

    // error
```

```

        // Test() { testConst = 1; }

        // error
        // Test(){
        const int testConst;
    };

    // error
    // int Test::testConst = 1;

    int main()
    {
        Test test;
        cout << test.testConst << endl;
        return 0;
    }

```

- 指针 this 是一个常量，因此 this 作为赋值、递增、递减、操作符的目的对象都是错误的

## 7.9 STATIC 修饰词

### 7.9.1 初始化时间

全局变量、文件域的静态变量和类的静态成员变量在 **main** 执行之前的静态初始化过程中分配内存并初始化；局部静态变量（一般为函数内的静态变量）在第一次使用时分配内存并初始化。这里的变量包含内置数据类型和自定义类型的对象。

非局部静态变量一般在 **main** 执行之前的静态初始化过程中分配内存并初始化，可以认为是线程安全的；

### 7.9.2 静态全局变量

(1) 该变量在全局数据区分配内存

(2) 未经初始化的静态全局变量会被程序自动初始化为 0（自动变量的值是随机的，除非它被显式初始化）

(3) 静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的

(4) 全局数据区的数据并不会因为函数的退出而释放空间

-> 程序的内存分布

1- 代码区

2- 全局数据区: 静态数据（即使是函数内部的静态局部变量）也存放在全局数据区

3- 堆区: 由 new 产生的动态数据存放在堆区

4- 栈区: 函数内部的自动变量存放在栈区, 自动变量一般会随着函数的退出而释放空间

### 7.9.3 静态局部变量

(1) 在函数体内定义了一个变量，每当程序运行到该语句时都会给该局部变量分配栈内存。但随着程序退出函数体，系统就会收回栈内存，局部变量也相应失效。但有时候我们需要在两次调用之间对变量的值进行保存。通常的想法是定义一个全局变量来实现。但这样一来，变量已经不再属于函数本身了，不再仅受函数的控制，给程序的维护带来不便。

(2) 该变量在全局数据区分配内存

(3) 静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化

(4) 静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0

(5) 它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束

### 7.9.4 静态函数

(1) 在函数的返回类型前加上 `static` 关键字，函数即被定义为静态函数。静态函数与普通函数不同，它只能在声明它的文件当中可见，不能被其它文件使用

(2) 静态函数不能被其它文件所用

(3) 其它文件中可以定义相同名字的函数，不会发生冲突

### 7.9.5 静态数据成员

(1) 在类内数据成员的声明前加上关键字 `static`，该数据成员就是类内的静态数据成员

(2) 对于非静态数据成员，每个类对象都有自己的拷贝。而静态数据成员被当作是类的成员。无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份拷贝，由该类型的所有对象共享访问

(3) 静态数据成员存储在全局数据区。静态数据成员定义时要分配空间，所以不能在类声明中定义或初始化，同其他一样在 `cpp` 文件中初始化

(4) 静态数据成员和普通数据成员一样遵从 `public, protected, private` 访问规则

### 7.9.6 静态成员函数

(1) 静态成员函数与静态数据成员一样，都是类的内部实现，属于类定义的一部分。普通的成员函数一般都隐含了一个 `this` 指针，`this` 指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的，但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有 `this` 指针，从这个意义上讲，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数。

(2) 出现在类体外的函数定义不能指定关键字`static`, 即声明函数时加`static` , 在外面实现时不加

(3) 由于没有`this` 指针的额外开销, 因此静态成员函数与类的全局函数相比速度上会有少许的增长



# 第八章 this

## 8.1 概念

this 是 C++ 中的一个关键字，也是一个 **const** 指针，它指向当前对象，通过它可以访问当前对象的所有成员。

所谓当前对象，是指正在使用的对象。例如对于 `stu.show()`，`stu` 就是当前对象，`this` 就指向 `stu`。

```
void Student::printThis(){
    cout<<this<<endl;
}
Student *pstu1 = new Student;
pstu1 -> printThis();
cout<<pstu1<<endl;

Student *pstu2 = new Student;
pstu2 -> printThis();
cout<<pstu2<<endl;
```

运行结果：

```
0x7b17d8
0x7b17d8
0x7b17f0
0x7b17f0
```

## 8.2 注意问题

<https://www.cnblogs.com/lijia0511/p/4936843.html>

- `this` 是 `const` 指针，它的值是不能被修改的，一切企图修改该指针的操作，如赋值、递增、递减等都是不允许的。
- `this` 只能在成员函数内部使用，用在其他地方没有意义，也是非法的。
- 只有当对象被创建后 `this` 才有意义，因此不能在 `static` 成员函数、全局函数中使用。
- `this` 指针本质是一个函数参数，只是编译器隐藏起形式的，语法层面上的参数。实际上，成员函数默认的第二个参数为 `T* const this`

- *this* 在成员函数的开始前构造，在成员的结束后消除。这个生命周期同任何函数的参数是一样的，没有任何区别。当调用一个类的成员函数时，编译器将类的指针作为函数的 *this* 参数传递进去。
- *this* 指针并不占用对象的空间。所有成员函数的参数，不管是不是隐含的，都不会占用对象的空间，只会占用参数传递时的栈空间，或者直接占用一个寄存器。



# 第九章 继承

## 9.1 潜规则

(1) 不指明继承方式关键字`public`时，编译器会默认继承方式为`private`或`protected`

(2) 基类的所有私有成员仅在基类中可见，而在派生类中是不可见的。基类的私有成员可以由派生类继承，但是派生类不可见

(3) 继承相当于把父类所有的东西复制了一份，但是可能因访问权限的设置而不能使用。

(4) 公有继承的保护成员，只能在派生类中访问，不能用派生类对象访问

(5) 如果派生类添加了一个数据成员，而该成员与基类中的某个数据成员同名，新的数据类型就隐藏了继承来的同名成员；同理，函数也存在隐藏

(6) 派生类可对从基类继承来的保护成员进行访问，也就是说保护成员在派生类中是可见的，但是只能在对象内部进行调用，但是不能通过对象在外部调用

例子：

```
class BC
{
public:
    void set_x(int a){x=a;}

protected:
    int get_x() const {return x;}
private:
    int x;
};

class DC:public BC
{
public:
    void add2() {int c = get_x(); set_x(c+2);} //Right Can read protected variable
};

int main()
{
    DC d;
```

```

    d.set_x(3);
    //cout << d.get_x() << endl; //Wrong! can not run,use the protected method out
    Object
    // d.x = 77; //Wrong! can not run
    d.add2();

    return 0;
}

```

main 函数中能够访问 DC 的公有成员 set\_x 和 add2，但不能访问保护成员 get\_x, get\_x 仅在类层次结构中可见（不能以对象的方式访问）

(7) 除了 friend 函数，只有处于类层次结构中的成员函数才能访问保护成员 [派生]

(8) 应避免将数据成员设计为保护类型，即使某个数据成员可以成为保护成员，但更好的解决方案是：首先将这个数据成员定义为私有成员，然后为它设计一个用来进行存取访问的保护成员函数

(9) 可以使用基类指针存贮子类，但是不能调用子类相关的函数：

```

class Super
{
public:
    Super();
    void someMethod();
protected:
    int mProtectedInt;
private:
    int mPrivateInt;
};

class Sub : public Super
{
public:
    Sub();
    void someOtherMethod();
};

Super mySuper;
mySuper.someOtherMethod(); // Error! Super doesn't have a someOtherMethod().

Super* superPointer = new Sub(); // Create sub, store it in super pointer.
superPointer->someOtherMethod(); // Error! However, you CanNot call methods from the
    Sub class through the Super pointer.

```

## 9.2 构造函数调用顺序

(1) 当创建一个派生类时，基类的构造函数被自动调用，用来对派生类对象中的基类部分初始化，并完成其他一些相关事务，但是必须在派生类的构造函数处明确优先调用，`:super(),you`

(2) 在类的层次结构中，构造函数按基类到派生类的次序执行，析构函数则按派生类到基类的次序执行

## 9.3 继承方式

### 9.3.1 继承后对父类资源的访问权限

各种继承后，派生类对父类资源的访问权限变为如图所示：

		继承方式			在派生类中
		public	protected	private	
基类成员	public	public	protected	private	
	protected	protected	protected	private	
	private	不可用	不可用	不可用	

### 9.3.2 继承约束

#### 禁止继承-final

将类定义为 `final` 意味着该类将不再允许被继承.

```
class Super final
{
    // Omitted for brevity
};

// The following Sub class tries to inherit from the Super class, but this will result
// in a compiler error because Super is marked as final.
class Sub : public Super
{
    // Omitted for brevity
};
```

### 9.3.3 重写父类方法

**Only methods that are declared as virtual in the base class can be overridden properly by derived classes.**

If you wish to **provide a new definition** for `someMethod()` in the `Sub` class, you must first **add it to the class definition** for `Sub`,as follows:

```

class Sub : public Super
{
public:
    Sub();
    virtual void someMethod() override; // Overrides Super's someMethod()
    virtual void someOtherMethod();
};

void Sub::someMethod()
{
    cout << "This is Sub's version of someMethod()." << endl;
}

```

Once a method or destructor is **marked** as virtual, it will be virtual for all derived classes even if the virtual keyword is removed from derived classes

```

class Sub : public Super
{
public:
    Sub();
    void someMethod() override; // Overrides Super's someMethod()
};

```

### 9.3.4 切片

当用 **基类指针或引用**子类对象，此时可以调用 **基类存在的函数或成员**，但是不能调用子类独有的函数与数据成员，但当调用 **赋值操作符**进行构造对象，那么此时对象调用的函数 **将是基类原版本**，而非子类重写的版本

```

Super mySuper;
mySuper.someMethod(); // Calls Super's version of someMethod().

Sub mySub;
mySub.someMethod(); // Calls Sub's version of someMethod()

Sub mySub;
Super& ref = mySub;
ref.someMethod(); // Calls Sub's version of someMethod()

Sub mySub;
Super& ref = mySub;
mySub.someOtherMethod(); // This is fine.
ref.someOtherMethod(); // Error

Sub mySub;
Super assignedObject = mySub; // Assigns a Sub to a Super.
assignedObject.someMethod(); // Calls Super's version of someMethod()

```

Derived classes retain their overridden methods when referred to by base `class` pointers or references. They lose their uniqueness when cast to a base `class` object. The loss of overridden methods and derived `class` data is called slicing.

**Casting Up and Down** When upcasting, use a pointer or reference to the base class to avoid slicing.

```
Super mySuper = mySub; // SLICE!
Super& mySuper = mySub; // No slice!

void lessPresumptuous(Super* inSuper)
{
    // Use downcasting only when necessary and be sure to use a dynamic_cast.
    Sub* mySub = dynamic_cast<Sub*>(inSuper);
    if (mySub != nullptr) {
        // Proceed to access Sub methods on mySub.
    }
}
```

### 9.3.5 使用父类方法

```
class B: public A {
public:
    void func3(string prefix) {
        cout << prefix << "B::func3" << endl;
        A::func3(prefix + "  ");
    }
};
```

## 9.4 支持多继承

一个派生类可以拥有多个基类

(1) 多继承中，多个基类每个基类都要加上访问修饰符，缺省则默认为私有的

## 9.5 虚基类

使用虚基类能够在多重派生的过程中，使共有的基类部分在派生类中只有一个拷贝，这样就能解决二义性的错误。

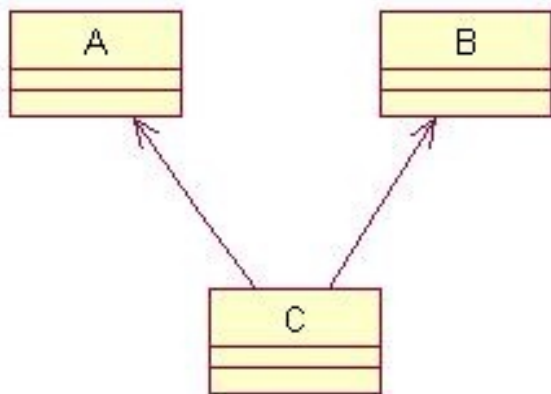


图 9.1: 普通多继承

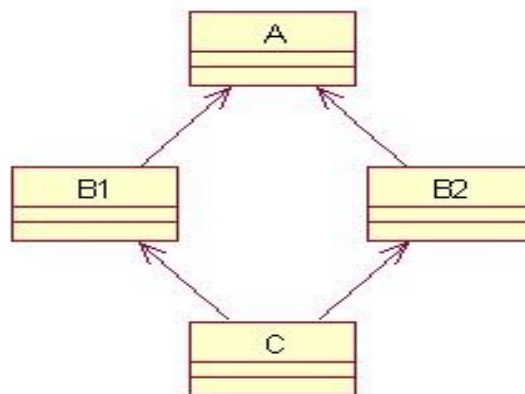


图 9.2: 二义性多继承

```

class A
{
public:
    A(int a = 0) : mA(a) {}
    int mA;
};

class B1 : virtual public A
{
public:
    B1(int a = 0, int b = 0) : A(b), mB1(a) {}
    int mB1;
};

class B2 : virtual public A
{
public:
    B2(int a = 0, int b = 0) : A(b), mB2(a) {}
    int mB2;
};

class C : public B1, public B2
{
public:
    C(int x, int a, int b, int c, int d) : B1(a, b), B2(c, d), mC(x) {}

    void Print() const
    {
        cout << "B1::mA_=" << B1::mA << endl;
        cout << "B2::mA_=" << B2::mA << endl;
        cout << "mA_=" << mA << endl;
        cout << "mB1_=" << mB1 << endl;
        cout << "mB2_=" << mB2 << endl;
    }
}
  
```

```

        cout << "mC_=__" << mC << endl;
    }

    int mC;
};

int main()
{
    C obj(10, 20, 30, 40, 50);
    obj.Print();
    cout << endl;
    obj.mA = 163;
    obj.Print();
    return 0;
}

```

-----Output----- :

```

B1::mA = 0
B2::mA = 0
mA = 0
mB1 = 20
mB2 = 40
mC = 10

B1::mA = 163
B2::mA = 163
mA = 163
mB1 = 20
mB2 = 40
mC = 10

```

(1) 共有部分为 mA; 私有部分为 mBx; 共有部分调用基类进行初始化 (没有显示调用基类)

(2) 虚基类的构造函数的调用方法与一般基类的构造函数的调用方法是不同的。在这个例子中，编译器没有调用 B1 或者 B2 的构造函数来调用基类 A 的构造函数，因为在虚继承过程中，基类 A 只有一个拷贝，所以编译器无法确定应该由类 B1 或者类 B2 的构造函数来调用基类 A 的构造函数，所以此时调用的是基类 A 的默认构造函数，所以刚开始 mA 的结果为 0，是基类 A 的默认构造函数设置的默认值。

(3) 由虚基类经过一次或者多次派生出来的派生类，在其每一个派生类的构造函数的成员初始化列表中必须给出对虚基类的构造函数的调用，如果未列出，则调用虚基类的默认构造函数。

(4) 在本例当中，在执行 B1 和 B2 的构造函数时都不调用虚基类 A 的构造函数，而是在类 C 中的构造函数直接调用虚基类 A 的默认构造函数。

(5) 如果将 C 的构造函数改为 C(int x, int a, int b, int c, int d) : A(a), B1(a, b),

B2(c, d), mC(x) , 则在这里显示地调用虚基类 A 的构造函数, 并传入初始值, 所以第一次打印 mA 的值不是 0, 而是 20。

```
-----Output-----  
  
B1::mA = 20  
B2::mA = 20  
mA = 20  
mB1 = 20  
mB2 = 40  
mC = 10  
  
B1::mA = 163  
B2::mA = 163  
mA = 163  
mB1 = 20  
mB2 = 40  
mC = 10
```

## 9.6 C++ 接口的实现方式

接口即只包含纯虚函数的抽象类

### 9.6.1 纯虚函数

纯虚函数是在基类中声明的虚函数, 它在基类中没有定义, 但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加 “=0”

`virtual 返回值类型成员函数名 (参数表) =0;`

### 9.6.2 抽象类

包含纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数, 所以不能定义抽象类的对象

重要的是抽象类可以包括抽象方法, 这是普通类所不能的, 但同时也能包括普通的方法。

抽象方法只能声明于抽象类中, 且不包含任何实现, 派生类必须覆盖它们

### 9.6.3 C++ 接口与抽象类的区别

<http://blog.csdn.net/hackbuteer1/article/details/7558946>



# 第十章 virtual 关键字

## 10.1 多态

实际上是一个函数指针，在运行期间把该函数指针指向子类的相同函数

**编译器绑定-非多态** 一个函数的名称和其入口地址是紧密相连的，入口地址是该函数在内存中的起始地址

由于函数被调用时，到底应该执行哪一段代码是由编译器在编译阶段就决定了的，因此我们将这种对函数的绑定方式称为编译器绑定 (compile-time binding): 专业术语：编译器将所以对函数的调用绑定到函数的入口地址

**运行期绑定-多态** 与编译器绑定不同的时，运行期绑定是直到程序运行之时，才将函数名称绑定到其入口地址。

如果对一个函数的绑定发生在运行期而非编译器，我们就称该函数是多态

**C++ 中多态有以下三个前提条件：**

1. 必须存在一个继承体系结构
2. 继承体系结构中的一些类型必须具有同名的 `virtual` 成员函数 (`virtual` 关键字)

3. 至少有一个基类类型的指针或基类类型的引用，这个指针或引用可用来对 `virtual` 成员函数进行调用

**\* 析构函数为什么要 `virtual` 的** 在公有继承中，基类对派生类及其对象的操作，只能影响到那些从基类继承下来的成员。如果想要用基类对非继承成员进行操作，则要把基类的这个函数定义为虚函数。

析构函数自然也应该如此：如果它想析构子类中的重新定义或新的成员及对象，当然也应该声明为虚的。

**占用空间问题** C++ 中虚函数的实现使用到了虚表，如果一个类的成员函数有虚函数，因为要存储虚表指针，其占用的空间会比存储所有数据成员所需要的内存空间大，从而造成与 C 语言中的结构体内存占用不一样，最终与 C 语言不兼容。

为了兼容 C 语言，C++ 实现的方式是对于所有无虚函数的类，都没有虚表指针，从而这样的类与 C 语言的结构体是可以兼容的。

一般是一个 int 的字节 (4);

深入 c++ 内存模型:<http://blog.csdn.net/qingyuanluofeng/article/details/48015901>

## 10.2 虚函数表

<http://www.cnblogs.com/hushpa/p/5707475.html#undefined>

**vtbl** 每一个 class 产生出一堆指向 virtual functions 的指针, 放在表格之中。这个表格被称为 virtual table(vtbl).

**vptr** 每个 class object 被安插一个指针, 指向相关的 virtual table. 通常这个指针被称为 vptr. vptr 的设置和重置都由每一个 class 的 constructor、destructor 和 copy assignment 运算符自动完成. 每个 class 所关联的 type\_info object(用以支持 runtime type identification, RTTI) 也经由 virtual table 被指出来, 通常该指针放在表格的第一个 slot.

虚函数(Virtual Function)是通过一张虚函数表(Virtual Table)来实现的。简称为V-Table。在这个表中, 主是要一个类的虚函数的地址表, 这张表解决了继承、覆盖的问题, 保证其容真实反应实际的函数。这样, 在有虚函数的类的实例中这个表被分配在了这个实例的内存中, 所以, 当我们用父类的指针来操作一个子类的时候, 这张虚函数表就显得尤为重要了, 它就像一个地图一样, 指明了实际所应该调用的函数。

C++ 的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置, 这意味着我们通过对象实例的地址得到这张虚函数表, 然后就可以遍历其中函数指针, 并调用相应的函数。

同属于一个类的对象共享虚函数表, 但是有各自的\_vptr

```
class Base {
public:
    virtual void f() {cout<<"base::f"<<endl;}
    virtual void g() {cout<<"base::g"<<endl;}
    virtual void h() {cout<<"base::h"<<endl;}
};

class Derive : public Base{
public:
    void g() {cout<<"derive::g"<<endl;}
};

//可以稍后再看
int main () {
    cout<<"sizeof_Base:"<<sizeof(Base)<<endl;

    typedef void(*Func)(void);
    Base b;
    Base *d = new Derive();

    long* pvptr = (long*)d;
```

```

    long* vptr = (long*)*pvptr;
    Func f = (Func)vptr[0];
    Func g = (Func)vptr[1];
    Func h = (Func)vptr[2];

    f();
    g();
    h();

    return 0;
}

```

```

/* Output */
size of Base: 8
base::f
derive::g
base::h

```

**new** 一个对象时，只为类中成员变量分配空间，对象之间共享成员函数。

运行下上面的代码发现sizeof(Base) == 8, 说明编译器在类中自动添加了一个 8 字节的成员变量，这个变量就是\_vptr, 指向虚函数表的指针。

1. d 对象的首地址就是vptr 指针的地址-pvptr
2. 取pvptr 的值就是vptr-虚函数表的地址
3. 取pvptr 的值等价于 \*d
4. 取vptr中[0][1][2] 的值就是这三个函数的地址，通过函数地址就直接可以运行三个虚函数了。
5. 函数表中Base::g() 函数指针被Derive 中的Derive::g() 函数指针覆盖，所以执行的时候是调用的Derive::g()

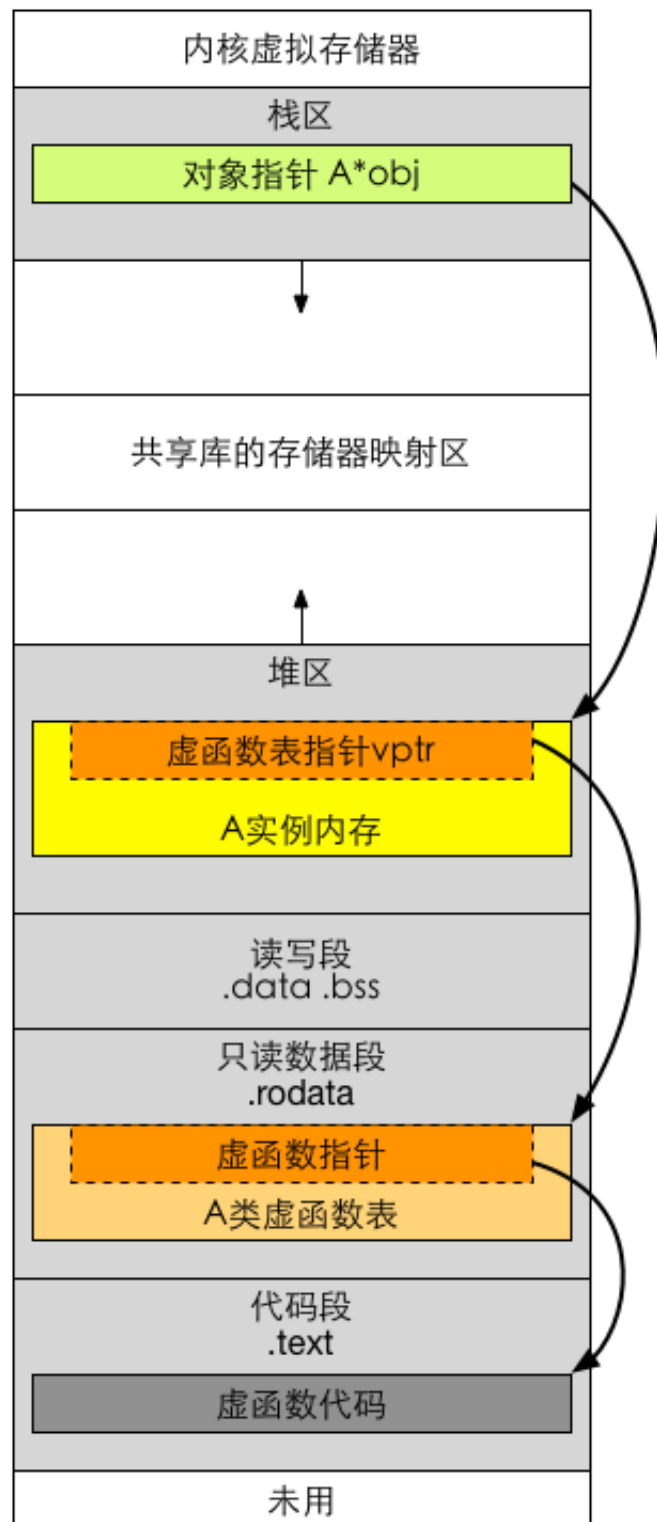


图 10.1: 进程虚拟内存图-对象结构

### 10.2.1 一般继承（无虚函数覆盖）

对于实例：Derive d; 的虚函数表如图10.2:

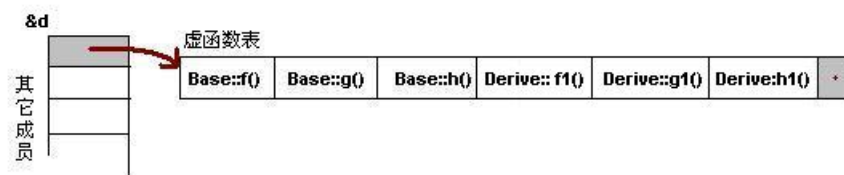


图 10.2: 一般继承（无虚函数覆盖）

- 虚函数按照其声明顺序放于表中
- 父类的虚函数在子类的虚函数前面

### 10.2.2 一般继承（有虚函数覆盖）

为了看到被继承过后的效果，在这个类的设计中，我只覆盖了父类的一个函数：`f()`。那么，对于派生类的实例，其虚函数表会是下面图10.3的一个样子：

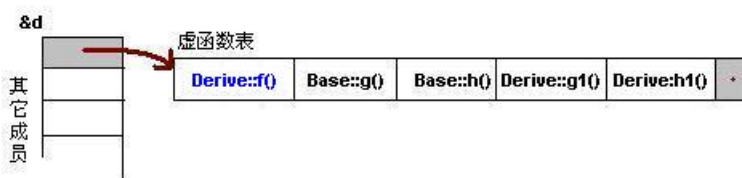


图 10.3: 一般继承（有虚函数覆盖）

- 覆盖的 `f()` 函数被放到了虚表中原来父类虚函数的位置
- 没有被覆盖的函数依旧

这样，我们就可以看到对于下面这样的程序 `Base *b = new Derive(); b->f();` 由 `b` 所指的内存中的虚函数表的 `f()` 的位置已经被 `Derive::f()` 函数地址所取代，于是在实际调用发生时，是 `Derive::f()` 被调用了。这就实现了多态。



# 第十一章 C++ 内存结构

## 11.1 进程结构

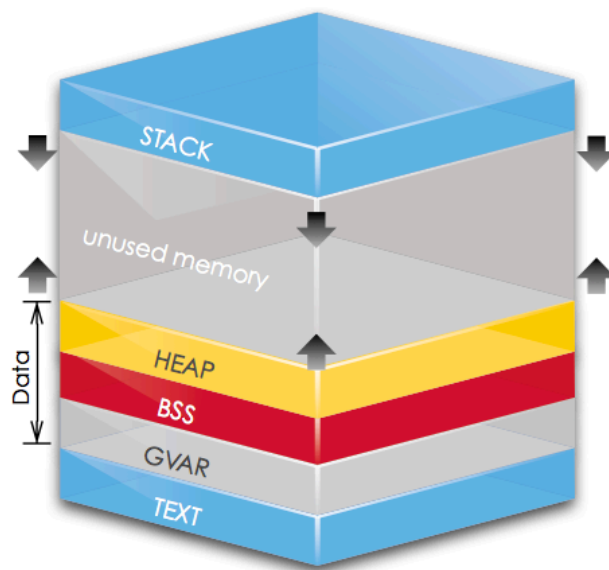


图 11.1: 进程内存结构

### 11.1.1 Stack 栈

栈，存放Automatic Variables，按内存地址由高到低方向生长，其最大大小由编译时确定，速度快，但自由性差，最大空间不大。

### 11.1.2 Heap 堆

堆，自由申请的空间，按内存地址由低到高方向生长，其大小由系统内存/虚拟内存上限决定，速度较慢，但自由性大，可用空间大。

每个线程都会有自己的栈，但是堆空间是共用的。

堆和栈都是动态分配内存，两者空间大小都是可变的。

### 11.1.3 .bss 未初始化全局与静态变量

存放程序中未初始化的和零值全局变量与静态变量。静态分配，在程序开始时通常会被清零。

### 11.1.4 .data 初始化的全局与静态变量

用来存放程序中已经初始化的非零全局变量与静态变量。静态分配 data 又可分为读写（RW）区域和只读（RO）区域。

- RO 段保存常量所以也被称为 `.constdata`
- RW 段则是普通非常全局变量，静态变量就在其中

### 11.1.5 .text 代码区

也称为代码段 (Code)，用来存放程序执行代码，同时也可能会包含一些常量 (如一些字符串常量等)。该段内存为静态分配，只读 (某些架构可能允许修改)。

这块内存是共享的，当有多个相同进程 (Process) 存在时，共用同一个 `.text` 段。

`text` 和 `data` 段都在可执行文件中，由系统从可执行文件中加载；而 `bss` 段不在可执行文件中，由系统初始化。这三段内存就组成了我们编写的程序的主体

## 11.2 对象结构

<http://www.cnblogs.com/gtarcoder/p/4929927.html>

<http://www.cnblogs.com/jerry19880126/p/3616999.html>

### 11.2.1 虚函数在内存中分布

对于有虚函数的基类和子类来说，内存中类对象的开始位置就是 4 字节的 `VPTR`（虚函数表的地址，指向 `VTABLE`）。

如果一个子类继承自基类，且没有对虚函数进行重写，子类仍然会自己维护一个虚函数表，和基类的虚函数表地址不同（尽管可能内容相同）。

可以这样理解，每个类都是一个架构模型，不同类型的实例则是需要根据不同的架构进行填充内存的。所以每个类都需要有各自的架构模型，如果是继承那么会将父类的模型拷贝过来再改造。

虚函数表中存放该类对应的实际会调用的函数地址，即若子类没有覆盖基类的虚函数，则对应位置存放基类虚函数地址；若子类覆盖了基类的虚函数，则对应位置存放子类的虚函数地址。虚函数表以 `NULL` 结尾。且如果子类中添加了基类中没有的虚函数，则新加的虚函数被放在子类虚函数表最后。

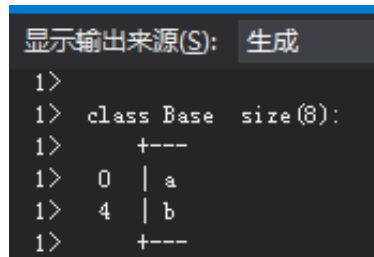
虚函数表只存放虚函数的地址，非虚函数由编译器在编译期间静态设定。

如果一个子类是多继承，即含有多个基类，则有多组虚函数表，分别对应到不同的继承。



## 11.2.2 单一对象

```
class Base
{
    int a;
    int b;
public:
    void CommonFunction();
};
```



```
显示输出来源(S): 生成
1>
1> class Base size(8):
1> +---
1> 0 | a
1> 4 | b
1> +---
```

图 11.2: 单一对象

c++ 类中有四种成员：

1. 非静态数据成员：放在每个对象内部，作为对象专有的数据成员
2. 静态数据成员：被抽取出来放在程序的静态数据区内，为该**类所有对象共享**，只保留一份
3. 非静态成员函数：
4. 静态成员函数：都被提取出来放在程序的代码段中并为该**类所有对象共享**，因此每个成员函数也只能存在一份代码实体

因此，构成对象的只有数据，任何成员函数都不属于任何一个对象，非静态成员函数和对象的关系就像是绑定，绑定的中介就是 `this` 指针。而 `this` 指针是以默认的类型指针对象作为第一个参数传进成员函数的，所以不占空间。

对象大小 = 非静态数据成员

### 11.2.3 单一对象 (virtual)

```
class Base
{
    int a;
    int b;
public:
    void CommonFunction();
    void virtual VirtualFunction();
};
```

```
class Base  size(12):
+---
0 | {vfptr}
4 | a
8 | b
+---

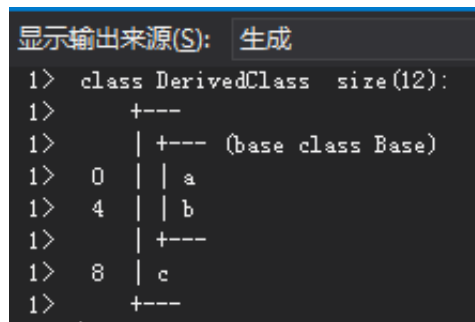
Base::$vftable@:
| @Base_meta
| 0
0 | @Base::VirtualFunction

Base::VirtualFunction this adjustor: 0
```

图 11.3: 单对象，包含虚函数

### 11.2.4 单继承 (base 无 virtual)

```
class DerivedClass: public Base
{
    int c;
public:
    void DerivedCommonFunction();
};
```



```
1> class DerivedClass size(12):
1> +---
1> | +--- (base class Base)
1> 0 | | a
1> 4 | | b
1> | +---
1> 8 | c
1> +---
```

图 11.4: 单继承

### 11.2.5 单继承 (base 有 virtual)

DerivedClass 继承了 Base，内存排布是先父类后子类。

```
class DerivedClass: public Base
{
    int c;
public:
    void DerivedCommonFunction();
    void virtual VirtualFunction();
};
```

```
> class DerivedClass size(16):
> +---
> | +--- (base class Base)
> 0 | | {vfptr}
> 4 | | a
> 8 | | b
> | +---
> 12 | c
> +---
>
> DerivedClass::$vftable@:
> | @DerivedClass_meta
> | 0
> 0 | @DerivedClass::VirtualFunction
>
> DerivedClass::VirtualFunction this adjustor: 0
```



图 11.5: 单继承，包含虚函数

### 11.2.6 虚继承-单继承 (virtual)

```
class DerivedClass1: virtual public Base
{
    int c;
public:
    void DerivedCommonFunction();
    void virtual VirtualFunction();
};

class DerivedClass2 : virtual public Base
{
    int d;
public:
    void DerivedCommonFunction();
    void virtual VirtualFunction();
};

class DerivedDerivedClass : public DerivedClass1, public DerivedClass2
{
    int e;
public:
    void DerivedDerivedCommonFunction();
    void virtual VirtualFunction();
};
```

```
class DerivedClass1 size(20):
+---
0  | {vbptr}
4  | c
+---
+--- (virtual base Base)
8  | {vfptr}
12 | a
16 | b
+---
```

图 11.6: 虚单继承，包含虚函数

### 11.2.7 多继承 (base 有 virtual)

如果一个子类有多个基类，为多继承。

其对象在内存中的布局为：

1. 按照多个继承，分为多个块；
2. 每块的开头为一种继承的虚函数表，接着为该种继承的基类的非静态成员变量
3. 所有块结束之后，为该类特有的非静态成员变量
4. 如果该类又新加了虚函数，则虚函数放在第一个虚函数表的最后。

```
// 显示菱形继承
class Base
{
    int a;
    int b;
public:
    void CommonFunction();
    void virtual VirtualFunction();
};

class DerivedClass1: public Base
{
    int c;
public:
    void DerivedCommonFunction();
    void virtual VirtualFunction();
};

class DerivedClass2 : public Base
{
    int d;
public:
    void DerivedCommonFunction();
    void virtual VirtualFunction();
};

class DerivedDerivedClass : public DerivedClass1, public DerivedClass2
{
    int e;
public:
    void DerivedDerivedCommonFunction();
    void virtual VirtualFunction();
};
```

// 验证子类新建虚函数位置 添加在 第一张虚表上

```
class A{
    public:
    virtual void f1(){
        cout << "f1_in_class_A" << endl;
    }
    private:
    int a;
};
class B{
    public:
    virtual void f2(){
        cout << "f2_in_class_B" << endl;
    }
    virtual void f4(){
        cout << "f4_in_class_B" << endl;
    }
    int b;
}
class D:public A, B{
    public:
    void f1(){
        cout << "f1_in_class_D" << endl;
    }
    void f2(){
        cout << "f2_in_class_D" << endl;
    }
    virtual void f3(){
        cout << "f3_in_class_D" << endl;
    }
    private:
    double c;
}
```



图 11.7: 多继承，包含虚函数

由外向内看，它并列地排布着继承而来的两个父类 `DerivedClass1` 与 `DerivedClass2`，还有自身的成员变量 `e`。`DerivedClass1` 包含了它的成员变量 `c`，以及 `Base`，`Base` 有一个 0 地址偏移的虚表指针，然后是成员变量 `a` 和 `b`；`DerivedClass2` 的内存排布类似于 `DerivedClass1`，注意到 `DerivedClass2` 里面竟然也有一份 `Base`。

**菱形继承** 如果有多个类继承自同一个基类，最后又被同一个子类继承这些类（例如 `B`, `C` 继承自 `A`，`D` 继承自 `B`, `C`），这种情况下，若类型 `A` 存在非静态成员变量 `a`，则会有 `D` 类型的对象保留从 `B` 继承来的 `a` 和从 `C` 继承来的 `a`，出现空间冗余。

使用虚继承，可以使得 `D` 从 `B` 继承来的 `a` 和从 `C` 继承来的 `a` 是同一个，节省空间。



### 11.2.8 虚继承-多继承 (base 有 virtual)

```
class Base
{
    int a;
    int b;
public:
    void CommonFunction();
    void virtual VirtualFunction();
};

class DerivedClass1: virtual public Base
{
    int c;
public:
    void DerivedCommonFunction();
    void virtual VirtualFunction();
};

class DerivedClass2 : virtual public Base
{
    int d;
public:
    void DerivedCommonFunction();
    void virtual VirtualFunction();
};

class DerivedDerivedClass : public DerivedClass1, public DerivedClass2
{
    int e;
public:
    void DerivedDerivedCommonFunction();
    void virtual VirtualFunction();
};
```

**Derived 类图** DerivedClass1 就已经有变化了，原来是先排虚表指针与Base 成员变量，vfptr 位于 0 地址偏移处；但现在有两个虚表指针了，一个是vbptr，另一个是vfptr。vbptr 是这个DerivedClass1 对应的虚表指针，它指向DerivedClass1 的虚表vbtable，另一个vfptr 是虚基类表对应的虚指针，它指向vftable。

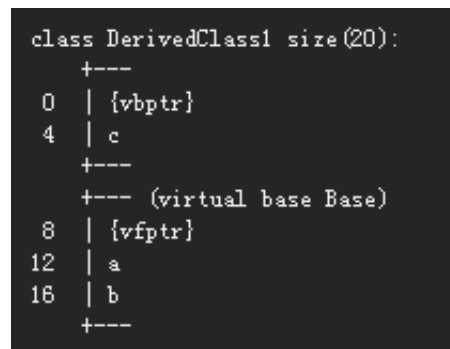


图 11.8: 虚多继承-D

**DerivedDerived 类图** 这里面有三个虚指针了，但base 却只有一份。第一张虚表是内含DerivedClass1 的，第二张虚表是内含DerivedClass2 的，最后一张表是虚基表。

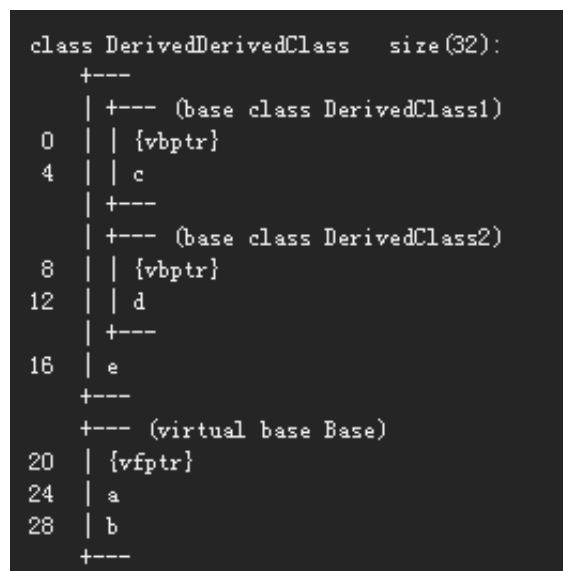


图 11.9: 虚多继承-DD

总结 ->

虚继承的作用是减少了对基类的重复，代价是增加了虚表指针的负担（更多的虚表指针）。

如果是虚继承，那么子类会有两份虚指针，一份指向自己的虚表，另一份指向虚基表，多重继承时虚基表与虚基表指针有且只有一份

## 11.3 ELF

<https://www.cnblogs.com/gtarcoder/p/6006023.html>

<http://www.cnblogs.com/xmphenix/archive/2011/10/23/2221879.html>



## 第十二章 extern 关键字

### 12.1 基本解释

extern 可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义

### 12.2 问题

作用范围-可见性的关键字

**extern 变量** 该关键字告诉编译器，其声明的函数和变量可以在本模块或其它模块中使用。

在一个源文件里定义了一个数组：char a[6]; 在另外一个文件里用下列语句进行了声明：

```
extern char *a;
```

请问，这样可以吗？

答案与分析：

答：不可以，程序运行时会告诉你非法访问。原因在于，指向类型 T 的指针并不等价于类型 T 的数组。extern char \*a 声明的是一个指针变量而不是字符数组，因此与实际的定义不同，从而造成运行时非法访问。应该将声明改为extern char a[ ]。

注意：你在 \*.c 文件中声明了一个全局的变量，这个全局的变量如果要被引用，就放在 \*.h 中并用 extern 来声明

连接申明 linkage declaration

**extern "C"** 被extern "C" 修饰的变量和函数是按照C 语言方式编译和连接的

在C++ 环境下使用C 函数的时候，常常会出现编译器无法找到obj 模块中的C 函数定义，从而导致链接失败的情况，应该如何解决这种情况呢？

答案与分析：

C++ 语言在编译的时候为了解决函数的多态问题，会将函数名和参数联合起来生成一个中间的函数名称，而C 语言则不会。

因此会造成链接时找不到对应函数的情况，此时 C 函数就需要用extern "C" 进行链接指定，这告诉编译器，请保持我的名称，不要给我生成用于链接的中间函数名。下面是一个标准的写法[函数的头文件-在.h 文件]：

```
#ifdef __cplusplus
```

```
#if __cplusplus
    extern "C"{
#endif
#endif /* __cplusplus */

/*
 *
 * defined what u use
 *
 */

#ifdef __cplusplus
    #if __cplusplus
    }
#endif
#endif /* __cplusplus */
```

**extern 函数声明** 当函数提供方单方面修改函数原型时，如果使用方不知情继续沿用原来的extern 申明，这样编译时编译器不会报错。但是在运行过程中，因为少了或者多了输入参数，往往会照成系统错误，这种情况应该如何解决？

答案与分析：

目前业界针对这种情况的处理没有一个很完美的方案,通常的做法是提供方在自己的xxx\_pub.h 中提供对外部接口的声明，然后调用方include 该头文件，从而省去extern 这一步。以避免这种错误。宝剑有双锋，对extern 的应用，不同的场合应该选择不同的做法。

# 第十三章 volatile 关键字

## 13.1 含义

1. 不会在两个操作之间把 volatile 变量缓存在寄存器中。在多任务、中断、甚至 setjmp 环境下，变量可能被其他的程序改变，编译器自己无法知道，volatile 就是告诉编译器这种情况。
2. 不做常量合并、常量传播等优化
3. 对 volatile 变量的读写不会被优化掉。如果你对一个变量赋值但后面没用到，编译器常常可以省略那个赋值操作，然而对 Memory Mapped IO 的处理是不能这样优化的。

## 13.2 示例

volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

声明时语法 : `int volatile vInt;`

**volatile 变量调用过程** 当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。例如：

```
int volatile i = 10;
int    a = i;
...
// 其他代码，并未明确的告诉编译器，对i 进行过操作。
int    b = i;
```

volatile 指出 *i* 是随时可能发生变化的，每次使用它的时候必须从 *i* 的地址中读取，因而编译器生成的汇编代码会重新从 *i* 的地址读取数据放在 *b* 中。

而优化做法是，由于编译器发现两次从 *i* 读数据的代码之间的代码没有对 *i* 进行过操作，它会自动把上次读的数据放在 *b* 中。而不是重新从 *i* 里面读。这样以来，如果 *i* 是一个寄存器变量或者表示一个端口数据就容易出错，所以说 volatile 可以保证对特殊地址的稳定访问。

注意，在 VC6 中，一般调试模式没有进行代码优化，所以这个关键字的作用看不出来。下面通过插入汇编代码，测试有无 volatile 关键字，对程序最终代码的影响：

```

int i = 10;
int a = i;
cout << "i_=" << a << endl;
...
// 其他代码，并未明确的告诉编译器，对i 进行过操作。
// 下面的汇编语句的作用就是改变内存中i 的值，但是又不让编译器知道
__asm{
    mov dword ptr [ebp-4], 20h
}

int b = i;
cout << "i_=" << b << endl;

// 在DEBUG 模式下输出为
i = 10
i = 32

// 在RELEASE 模式下输出为
i = 10
i = 10

```

输出的结果明显表明，Release 模式下，编译器对代码进行了优化，第二次没有输出正确的 i 值。下面，我们把 i 的声明加上 volatile 关键字 `int volatile i = 10;`，此时则不论 DEBUG 模式还是 RELEASE 模式都是没有优化的结果

```

int volatile i = 10;
int a = i;
cout << "i_=" << a << endl;
...
// 其他代码，并未明确的告诉编译器，对i 进行过操作。
// 下面的汇编语句的作用就是改变内存中i 的值，但是又不让编译器知道
__asm{
    mov dword ptr [ebp-4], 20h
}

int b = i;
cout << "i_=" << b << endl;

// 在DEBUG 模式下输出为
i = 10
i = 32

// 在RELEASE 模式下输出为
i = 10
i = 32

```



这说明这个 `volatile` 关键字发挥了它的作用。其实不只是“内嵌汇编操纵栈”这种方式属于编译无法识别的变量改变，另外更多的可能是多线程并发访问共享变量时，一个线程改变了变量的值，怎样让改变后的值对其它线程 `visible`。一般说来，`volatile` 用在如下的几个地方：

1. 中断服务程序中修改的供其它程序检测的变量需要加`volatile`
2. 多任务环境下各任务间共享的标志应该加`volatile`
3. 存储器映射的硬件寄存器通常也要加`volatile` 说明，因为每次对它的读写都可能由不同意义

## 13.3 volatile 指针

## 13.4 多线程下的 volatile

有些变量是用 `volatile` 关键字声明的。当两个线程都要用到某一个变量且该变量的值会被改变时，应该用 `volatile` 声明，该关键字的作用是防止优化编译器把变量从内存装入 CPU 寄存器中。如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造成程序的错误执行。`volatile` 的意思是让编译器每次操作该变量时一定要从内存中真正取出，而不是使用已经存在寄存器中的值，如下：

```
volatile BOOL bStop = FALSE;
```

在一个线程中：

```
while( !bStop )
{
    ...
}

bStop = FALSE;
return;
```

在另外一个线程中 要终止上面的线程循环：

```
bStop = TRUE;
while( bStop );
```

等待上面的线程终止，如果 不使用`volatile` 申明，那么这个循环将是一个死循环，因为 已经读取到了寄存器中，寄存器中 的值永远不会变成**FALSE**，加上`volatile`，程序在执行时，每次均从内存中读出 的值，就不会死循环了。

这个关键字是用来设定某个对象的存储位置在内存中，而不是寄存器中。因为一般的对象编译器可能会将其的拷贝放在寄存器中用以加快指令的执行速度，例如下段代码中：

```
...  
int nMyCounter = 0;  
for(; nMyCounter<100;nMyCounter++)  
{  
    ...  
}  
...
```

在此段代码中, `nMyCounter` 的拷贝可能存放到某个寄存器中(循环中, 对 `nMyCounter` 的测试及操作总是对此寄存器中的值进行), 但是另外又有段代码执行了这样的操作: `nMyCounter -= 1;` 这个操作中, 对 `nMyCounter` 的改变是对内存中的 `nMyCounter` 进行操作, 于是出现了这样一个现象: `nMyCounter` 的改变不同步。

## 13.5 参考

基本: [http://www.cnblogs.com/yc\\_sunniwell/archive/2010/06/24/1764231.html](http://www.cnblogs.com/yc_sunniwell/archive/2010/06/24/1764231.html)

详细: [https://www.cnblogs.com/yc\\_sunniwell/archive/2010/07/14/1777432.html](https://www.cnblogs.com/yc_sunniwell/archive/2010/07/14/1777432.html)

## 第十四章 explicit 关键字

### 14.1 作用

explicit 关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换-根据构造函数，只能以显示的方式进行类型转换。

explicit 关键字作用于单个参数的构造函数。

示例：

```
//////////未加explicit时的隐式类型转换//////////
class Circle
{
public:
    Circle(double r) : R(r) {}
    Circle(int x, int y = 0) : X(x), Y(y) {}
    Circle(const Circle& c) : R(c.R), X(c.X), Y(c.Y) {}
private:
    double R;
    int X;
    int Y;
};

int _tmain(int argc, _TCHAR* argv[])
{
    //发生隐式类型转换
    //编译器会将它变成如下代码
    //tmp = Circle(1.23)
    //Circle A(tmp);
    //tmp.~Circle();
    Circle A = 1.23;
    //注意是int型的，调用的是Circle(int x, int y = 0)
    //它虽然有2个参数，但后一个有默认值，任然能发生隐式转换
    Circle B = 123;
    //这个算隐式调用了拷贝构造函数
    Circle C = A;

    return 0;
}
```

////////加了explicit关键字后，可防止以上隐式类型转换发生////////

```
class Circle
{
public:
    explicit Circle(double r) : R(r) {}
    explicit Circle(int x, int y = 0) : X(x), Y(y) {}
    explicit Circle(const Circle& c) : R(c.R), X(c.X), Y(c.Y) {}
private:
    double R;
    int X;
    int Y;
};

int _tmain(int argc, _TCHAR* argv[])
{
    //一下3句，都会报错
    //Circle A = 1.23;
    //Circle B = 123;
    //Circle C = A;

    //只能用显示的方式调用了
    //未给拷贝构造函数加explicit之前可以这样
    Circle A = Circle(1.23);
    Circle B = Circle(123);
    Circle C = A;

    //给拷贝构造函数加了explicit后只能这样了
    Circle A(1.23);
    Circle B(123);
    Circle C(A);
    return 0;
}
```

# 第十五章 friend 关键字

## 15.1 目的

采用类的机制后实现了数据的隐藏与封装，只有类的成员函数才能访问类的私有成员，程序中的其他函数是无法访问私有成员的。

非成员函数可以访问类中的公有成员，但是如果将数据成员都定义为公有的，这又破坏了隐藏的特性。另外，应该看到在某些情况下，需要频繁地访问类的数据成员，特别是在对某些成员函数多次调用时，由于参数传递，类型检查 and 安全性检查等都需要时间开销，而影响程序的运行效率。这时可以将这些函数定义为该函数的友元函数

友元函数是可以直接访问类的私有成员的非成员函数。它是定义在类外的普通函数，但需要在类的定义中加以声明，声明时只需在友元的名称前加上关键字friend

## 15.2 友元函数

```
class CPoint
{
public:
    CPoint();
    CPoint(double _x, double _y){ x = _x; y=_y;}
    friend double calcDistance(CPoint& pt1, CPoint& pt2);
private:
    double x;
    double y;
};

double calcDistance(CPoint& pt1, CPoint& pt2)
{
    double dx = pt1.x - pt2.x;
    double dy = pt1.y - pt2.y;
    return sqrt(dx*dx+dy*dy);
}

int main(int argc, char* argv[])
{
    CPoint pt1(3.0, 4.0);
    CPoint pt2(4.0, 8.0);
```

```

double dist = calcDistance(pt1, pt2);
cout<<"calcDistance(pt1,pt2)="<<dist<<endl;
getchar();
return 1;
}

```

## 15.3 友元类

当一个类作为另一个类的友元时，这就意味着这个类的所有成员函数都是另一个类的友元函数

**例子**：在感情生活中，我们经常会遇到一些霸道的对象说：你的就是我的，我的就是我的（need is word ,word is still word）！，你有多少异性朋友，不能private，你银行卡的密码、QQ 密码需要不能private，即使private了，也要让我知道。呵呵，当然，对于一些大爱无私的男的，也总会满足自己女友的一些霸道条款。为了说明友元类，看下例

```

class CBoyFriend
{
public:
    CBoyFriend():girl_number(10),qq_password("123456789"){
        friend class CGirlFriend;

private:
    int howManyGirlFriends(){ return girl_number;}
    string whenToBuyRoses(){ return string("everyNight");}
private:
    string bank_card;
    string bank_card_password;
    string qq_id;
    string qq_password;
    int girl_number;
};

class CGirlFriend
{
public:
    CGirlFriend(){};
    CBoyFriend boy;
    void printBoyFriend()
    {
        cout<<"bank_card:"<<boy.bank_card<<endl;
        cout<<"password:"<<boy.bank_card_password<<endl;
        cout<<"qq_id:"<<boy.qq_id<<endl;
        cout<<"qq_password:"<<boy.qq_password<<endl;
    }
}

```

```
};

int main(int argc, char* argv[])
{
    CGirlFriend girl;
    girl.printBoyFriend();

    getchar();
    return 1;
}
```

(1) 友元函数不能被继承

(2) 友元关系是单向的，不具有交换性。若类 B 是类 A 的友元，类 A 不一定是类 B 的友元，要看在类中是否有相应的声明

(3) 友元关系不具有传递性。若类 B 是类 A 的友元，类 C 是 B 的友元，类 C 不一定是类 A 的友元





# 第十六章 mutable 关键字

## 16.1 目的

mutable 的中文意思是“可变的，易变的”，跟 constant（既 C++ 中的 const）是反义词。

在 C++ 中，mutable 也是为了突破 const 的限制而设置的。被 mutable 修饰的变量，将永远处于可变的狀態，即使在一个 const 函数中。

我们知道，如果类的成员函数不会改变对象的状态，那么这个成员函数一般会声明成 const 的。但是，有些时候，我们需要在 const 的函数里面修改一些跟类状态无关的数据成员，那么这个数据成员就应该被 mutable 来修饰。

## 16.2 示例

```
class ClxTest
{
public:
    ClxTest();
    ~ClxTest();

    void Output() const;
    int GetOutputTimes() const;

private:
    mutable int m_iTimes;
};

ClxTest::ClxTest()
{
    m_iTimes = 0;
}

ClxTest::~~ClxTest(){}

// const 函数修改与类状态无关的数据成员
void ClxTest::Output() const
{
    cout << "Output_for_test!" << endl;
```

```
        m_iTimes++;
    }

    int ClxTest::GetOutputTimes() const
    {
        return m_iTimes;
    }

    void OutputTest(const ClxTest& lx)
    {
        cout << lx.GetOutputTimes() << endl;
        lx.Output();
        cout << lx.GetOutputTimes() << endl;
    }
```

# 第十七章 类型转换

## 17.1 static\_cast

static\_cast: 基本拥有与 C 旧式转型相同的威力与意义, 以及相同的限制。它用的最多, 主要是在基本类型之间的转换

```
void test1()
{
    int first=23,second=31;
    double res=(double)first/second; //旧式C语法
    double res2=static_cast<double>(first)/second;//新式C++转型符
    cout<<res<<" " <<res2<<endl;// 0.741935 0.741935
    // char* str="789";
    // cout<<static_cast<int>(str)<<endl;//error: 无法从char*转换为int
}
```

## 17.2 const\_cast

const\_cast: 用来去掉表达式中的常量性 (constness), 常量属性多体现在指针和引用, 因为如果没有指针和引用, 就不存在不小心修改了不能修改的数据

## 17.3 dynamic\_cast

dynamic\_cast: 用来执行继承体系中“安全的向下转型或跨系转型动作”, 就是父类对象指针转化为子类对象指针。可以利用dynamic\_cast 将指向 base classObject 的 pointer 或 reference 转型为指向 derived classObject 的 pointer 或 reference, 如果转型失败, 会以一个 null 指针 (转换的是 pointer 的话) 或一个 exception 表现出来 (转换的是 reference 的话)

```
class B
{
public:
    virtual void fun()
    {
        cout<<"B.fun()"<<endl;
    }
};
```

```

class D1:public B
{
public:
    void fun()
    {
        cout<<"D1.fun()"<<endl;
    }
    void fun2()
    {
        cout<<"D1.fun2()"<<endl;
    }
};

class CBase{};

class CDerived : public CBase{};

void test3()
{
    // 父类指针转换为子类指针时，父类要有虚函数
    B* pb=new D1();
    D1* pd1=dynamic_cast<D1*>(pb);

    // 子类指针转换为父类指针，不需要虚函数
    CDerived dc;
    CDerived* dp=&dc;
    CBase* cc1=dp; // 老式：子类指针转换为父类指针，即父类指针指向子类对象
    CBase* cb1=dynamic_cast<CBase*>(dp); // 使用dynamic_cast将指向继承类的指针转化为指向
        基类的指针
    printf("point:_%d,_%d,_%d,_%d",dp,cc1,cb1,&dc); // 它们的地址相同，说明它们在内存中是
        同一个地址；即子类指针转换为父类指针，父类的地址指向子类的地址，且是相同的。

    CBase& cc2=dc;
    CBase& cb2=dynamic_cast<CBase&>(dc); // 使用dynamic_cast将指向继承类的引用转化为指向
        基类的引用
}

```

## 17.4 reinterpret\_cast

[http://www.cnblogs.com/ider/archive/2011/07/30/cpp\\_cast\\_operator\\_part3.html](http://www.cnblogs.com/ider/archive/2011/07/30/cpp_cast_operator_part3.html)

`reinterpret_cast <new_type> (expression)`

`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。

`reinterpret_cast` 可以，或者说应该在什么地方用来作为转换运算符：

- 从**指针类型**到一个**足够大的整数类型**
- 从**整数类型**或者**枚举类型**到**指针类型**
- 从一个指向函数的**指针**到另一个**不同类型**的指向函数的**指针**
- 从一个指向对象的**指针**到另一个**不同类型**的指向对象的**指针**
- 从一个指向类函数成员的**指针**到另一个指向**不同类型**的函数成员的**指针**
- 从一个指向类数据成员的**指针**到另一个指向**不同类型**的数据成员的**指针**

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。

（所谓”足够大的整数类型”，取决于操作系统的参数，如果是 32 位的操作系统，就需要整形（`int`）以上的；如果是 64 位的操作系统，则至少需要长整形（`long`）。具体大小可以通过 `sizeof` 运算符来查看）。



# 第十八章 `static_assert` 关键字

## 18.1 简介

其语法很简单：`static_assert`(常量表达式, 提示字符串)。

如果第一个参数常量表达式的值为真 (true 或者非零值), 那么`static_assert` 不做任何事情, 就像它不存在一样, 否则会产生一条编译错误, 错误位置就是该`static_assert` 语句所在行, 错误提示就是第二个参数提示字符串。

## 18.2 说明

- 使用`static_assert`, 我们可以在编译期间发现更多的错误, 用编译器来强制保证一些契约, 并帮助我们改善编译信息的可读性, 尤其是用于模板的时候
- `static_assert` 可以用在全局作用域中, 命名空间中, 类作用域中, 函数作用域中, 几乎可以不受限制的使用
- 编译器在遇到一个`static_assert` 语句时, 通常立刻将其第一个参数作为常量表达式进行演算, 但如果该常量表达式依赖于某些模板参数, 则延迟到模板实例化时再进行演算, 这就让检查模板参数成为了可能
- 性能方面, 由于是`static_assert` 编译期间断言, 不生成目标代码, 因此`static_assert` 不会造成任何运行期性能损失

## 18.3 范例

```
// 下面是一个来自MSDN的简单范例: 该static_assert用来确保编译仅在32位的平台上进行, 不支持
64位的平台, 该语句可以放在文件的开头处, 这样可以尽早检查, 以节省失败情况下的编译时间
static_assert(sizeof(void *) == 4, "64-bit_code_generation_is_not_supported.");

struct MyClass
{
    char m_value;
};
```

```

struct MyEmptyClass
{
    void func();
};

// 确保MyEmptyClass是一个空类（没有任何非静态成员变量，也没有虚函数）
static_assert(std::is_empty<MyEmptyClass>::value, "empty_class_needed");

//确保MyClass是一个非空类
static_assert(!std::is_empty<MyClass>::value, "non-empty_class_needed");

template <typename T, typename U, typename V>
class MyTemplate
{
    // 确保模板参数T是一个非空类
    static_assert(
        !std::is_empty<T>::value,
        "T_should_be_a_non-empty_class"
    );

    // 确保模板参数U是一个空类
    static_assert(
        std::is_empty<U>::value,
        "U_should_be_an_empty_class"
    );

    // 确保模板参数V是从std::allocator<T>直接或间接派生而来，
    // 或者V就是std::allocator<T>
    static_assert(
        std::is_base_of<std::allocator<T>, V>::value,
        "V_should_inherit_from_std::allocator<T>"
    );
};

// 仅当模板实例化时，MyTemplate里面的那三个static_assert才会真正被演算，
// 藉此检查模板参数是否符合期望
// template class MyTemplate<MyClass, MyEmptyClass, std::allocator<MyClass>>;
//通过这个例子我们可以看出来，static_assert可以很灵活的使用，通过构造适当的常量表达式，我
// 们可以检查很多东西。比如范例中std::is_empty和std::is_base_of都是C++新的标准库提供的
// type traits模板，我们使用这些模板可以检查很多类型信息。

```

## 18.4 参考

<http://www.cnblogs.com/lvdongjie/p/4489835.html>



# 第十九章 命名空间

## 19.1 namespace 别名

```
namespace BieMing = nameSpace'sName
```

## 19.2 namespace 扩充

```
namespace sameName{}
```

例子：

```
namespace runrunrunrun
{
    int a(10);
    char *str("gogogo");
    namespace run //namespace's QianTao
    {
        int a(9);
    }
}

namespace runrunrunrun //namespace's Expand
{
    int y(5);
    //int a(15);Redefine error
}

namespace r = runrunrunrun;//namespace's aliaName

void main132()
{
    std::cout << r::run::a << std::endl;//namespace can QianTao
    std::cout << r::y << std::endl;
    std::cout << r::a << std::endl;
}
```



## 第二十章 引用

### 20.1 引用创建时必须初始化

```
DataType & Value(referenceValue);  
DataType & Value = referenceValue;
```

### 20.2 引用不能重定义

即一旦定义，不能再重新指向别的变量。

一旦一个引用被初始化为指向一个对象，它就不能改变为另一个对象的引用，指针则可以在任何时候指向另一个对象。

### 20.3 函数中的引用

```
int* f(int* X)  
{  
    (*X)++;  
    return X; //Safe, X is outside this scope  
}  
  
int& g(int& x)  
{  
    x++; // Same effect as in f()  
    return x; // Safe, outside this scope  
}  
  
int& h()  
{  
    int q;  
    //return q; //Error, Because in the Stack Memory, Then we can only use once. if we  
    //use the pointer, Then if we save the address, Then we can use too, but not Safe  
    //at all. See Example 3.1  
    static int x;  
    return x; //Safe, x lives outside this scope.  
}
```

```

//Reference TO Pointer
void func(int* &i)
{
    i++; // change the int* pointer's address, not the value i.
}

```

### 例子 3.1 :

```

int& funcTest()
{
    int test = 10;
    return test;
}

int main()
{
    int& T = funcTest();
    /*if change to (int T) Then can work,
    Because we can catch when the top temp local value pop out(Stack),
    if such reference do not same the second.
    Because the refer first is a Temp Local Register value.
    second the Memory has been recalled ,so it is a gabash value.

    Heap:the C++ do not allow use the memory that have been deleted.

    int* p = new int;
    delete p;

    cout << p <<endl; That not allow
    */

    cout << T<<endl;
    cout << T<<endl;
    return 0;
}

```

## 20.4 引用右值

两个&, 即引用 CPU 的寄存器中的值或内存中的值

参考网址 <http://www.cnblogs.com/qicosmos/p/4283455.html>

右值引用的声明期限 getVar() 产生的临时值不会在表达式结束之后就销毁了, 而是会被“续命”, 他的生命周期将会通过右值引用得以延续, 和变量 k 的声明周期一样长

```
T&& k = getVar();
```

**类型不确定性** 仅仅是当发生自动类型推导（如函数模板的类型自动推导，或 auto 关键字）的时候，T&& 才是 universal references

**完美转发** 在函数模板中，完全依照模板的参数类型（即保持参数的左值、右值特征），将参数传递给函数模板中调用的另外一个函数



# 第二十一章 enum 的使用

## 21.1 定义

```
enum color:char{ red='A' , yellow, green, white };
enum class Enum2 : unsigned int {Val1, Val2};
enum Enum3 : unsigned long {Val1 = 1, Val2};
```

- 可以指定存储类型
- 可以指定存储值的开始，默认为 0（int，char 都是），本例为 ‘A’，则 yellow 为 B,...

## 21.2 使用

### 21.2.1 类外函数

```
color mycolor = red;
//mycolor = 'A';//确保在枚举的范围之内不出错，枚举不可以当成基本数据类型使用
mycolor = color::white;//新语法
color mycolor1(red);
color mycolor2(color::red);
```

### 21.2.2 类中

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    enum class Colors { Red = 1, Green, Blue, Yellow };
    void setColor(Colors color);
private:
    // Omitted for brevity
    Colors mColor = Colors::Red;
};

void SpreadsheetCell::setColor(Colors color)
{
```

```
        mColor = color;
    }

    SpreadsheetCell myCell(5);
    myCell.setColor(SpreadsheetCell::Colors::Blue);
```

## 21.3 注意事项

```
enum Enum1; // Illegal in C++03 and C++11; the underlying type cannot be determined.
enum Enum2 : unsigned int; // Legal in C++11, the underlying type is specified
                        explicitly.
enum class Enum3; // Legal in C++11, the underlying type is int.
enum class Enum4 : unsigned int; // Legal in C++11.
enum Enum2 : unsigned short; // Illegal in C++11, because Enum2 was formerly declared
                        with a different underlying type.
```



## 第二十二章 模版

参考 C++\_STL 中的泛型编程一章。



## 第二十三章 异常



# 第二十四章 编译

## 24.1 Windows-VisualStudio

### 24.1.1 配置全局 VC++ 目录

1. 随便打开一个项目，然后点击菜单中的 视图->其他窗口->属性管理器
2. 打开属性管理器，点击项目前的箭头，展开项目，找到Debug 或者Release 下面的 Microsoft.Cpp.Win32.user 这个属性
3. 双击会出现一个跟在项目上右键属性一样的窗口，修改里面的“VC++目录”就是修改了全局的，

### 24.1.2 配置第三方程序库-以 opencv 示例

下载配置所需

1.VisualStudio2010

2.OpenCV2.4.8

安装

1. 安装位置不影响

2. 配置 Path 系统环境     PATH 是路径的意思，PATH 环境变量中存放的值，就是一连串的路径。不同的路径之间，用英文的分号(;)分隔开。系统执行用户命令时，若用户未给出绝对路径，则首先在当前目录下寻找相应的可执行文件、批处理文件（另外一种可以执行的文件）等。若找不到，再依次在PATH 保存的这些路径中寻找相应的可执行的程序文件。系统就以第一次找到的为准；若搜寻完 PATH 保存的所有路径都未找到，则会显示类似于”找不到该命令”的错误信息。

配置

1.Create Project    建立一个 VisualStudio ConsoleApplication Empty Project

## 2. 配置

- -右键项目
- -Properties
- -VC++ Directories
- -配置 include Directories += 安装目录 \opencv\build\include; += 安装目录 \opencv\build\inc  
+= 安装目录 \opencv\build\include\opencv2;
- -配置 Library Directories += 安装目录 \opencv\build\x86\vc10\lib;
- -Linker 选项 -input
- Additional Dependencies 添加如下库：见参考文献

## 编程测试

注意：路径中的单杠要换成双杠\\

### 24.1.3 dll 与 lib 的配置

**静态 lib** 它将导出声明和实现均放到 lib 中，编译后所有代码都嵌入到宿主程序中去。

**动态 lib** 相当于一个 h 文件，它是对实现部分（.DLL）的导出部分的声明。编译后只是将导出声明部分编译到宿主程序中，运行时需要相应的 DLL 文件的支持，否则无法工作。当生成一个新的 DLL 时，也会有配套的 lib 产生（即二者需一起分发），此时的 lib 即为动态 lib->[如 OpenGL-glut 工具库]

动态库的 2 种配置方法：

1. 需把对应的.dll 文件以及.lib 文件和.h 文件（结合方式时）拷贝至调用的程序目录下
2. 在 visual studio 中把 c++ 目录的 include 目录和附属 lib 目录配置好后，再把对应的.dll 文件考到系统的 system32 或 sysWOW64 中.. 这样的好处是，一劳永逸。不过可能你并没有权限操作执行复制操作.. 此时参考下文。

### 24.1.4 常见错误

参考 dll 与 lib:<http://blog.csdn.net/heyabo/article/details/8721611>

win10 权限更改:<http://blog.163.com/qywwkai@126/blog/static/21003042201541235335362/>

## 24.2 Linux - Makefile

<http://blog.csdn.net/haoel/article/details/2887/>

### 24.2.1 Makefile 规则

```
target ... : prerequisites ...  
    command  
    ...  
    ...
```

- `target` 就是一个目标文件, 可以是Object File, 也可以是执行文件, 当然也可以是个标签(clean)
- `prerequisites` 就是要生成那个`target` 所需要的文件或是目标
- `command` 就是`make` 需要执行的命令-任意的Shell命令

#### 示例

```
edit : main.o kbd.o command.o display.o /  
      insert.o search.o files.o utils.o  
      cc -o edit main.o kbd.o command.o display.o /  
          insert.o search.o files.o utils.o  
main.o : main.c defs.h  
      cc -c main.c  
kbd.o : kbd.c defs.h command.h  
      cc -c kbd.c  
  
...  
  
utils.o : utils.c defs.h  
      cc -c utils.c  
clean :  
      rm edit main.o kbd.o command.o display.o /  
          insert.o search.o files.o utils.o
```

- 在定义好依赖关系后, 后续的那一行定义了如何生成目标文件的操作系统命令, 一定要以一个 `Tab` 键作为开头。
- 反斜杠 (`/`) 是换行符的意思

### 24.2.2 make 执行流程

1. `make` 会在当前目录下找名字叫“Makefile”或“makefile”的文件
2. 如果找到, 它会找文件中的第一个目标文件(`target`), 在上面的例子中, 他会找到“`edit`”这个文件, 并把这个文件作为最终的目标文件。
3. 如果`edit` 文件不存在, 或是 `edit` 所依赖的后面的 `.o` 文件的文件修改时间要比`edit` 这个文件新, 那么, 他就会执行后面所定义的命令来生成`edit` 这个文件

4. 如果edit 所依赖的.o 文件也存在, 那么make 会在当前文件中找目标为.o 文件的依赖性, 如果找到则再根据那一个规则生成.o 文件
5. 当然, 你的C 文件和H 文件是存在的啦, 于是make 会生成 .o 文件, 然后再用 .o 文件生命make 的终极任务, 也就是执行文件edit 了

### 24.2.3 makefile 中使用变量

```
objects = main.o kbd.o command.o display.o /  
         insert.o search.o files.o utils.o  
edit : $(objects)  
      cc -o edit $(objects)
```

### 24.2.4 make 的自动推倒

只要、ver 看到一个[.o]文件, 它就会自动的把[.c]文件加在依赖关系中, 如果make 找到一个whatever.o, 那么whatever.c, 就会是whatever.o 的依赖文件。并且 cc -c whatever.c 也会被推导出来

于是, 我们的makefile 再也不用写得这么复杂。我们的是新的makefile 又出炉了。

```
objects = main.o kbd.o command.o display.o /  
         insert.o search.o files.o utils.o  
  
edit : $(objects)  
      cc -o edit $(objects)  
  
main.o : defs.h  
kbd.o : defs.h command.h  
command.o : defs.h command.h  
display.o : defs.h buffer.h  
insert.o : defs.h buffer.h  
search.o : defs.h buffer.h  
files.o : defs.h buffer.h command.h  
utils.o : defs.h  
  
.PHONY : clean  
clean :  
      rm edit $(objects)
```

**进一步自动化** 即然我们的make 可以自动推导命令, 那么我看到那堆[.o]和[.h] 的依赖就有点不爽, 那么多的重复的[.h], 能不能把其收拢起来, 好吧, 没有问题, 这个对于make 来说很容易, 谁叫它提供了自动推导命令和文件的功能呢? 来看看最新风格的makefile 吧。

```
objects = main.o kbd.o command.o display.o /  
         insert.o search.o files.o utils.o
```



```
edit : $(objects)
    cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h

.PHONY : clean
clean :
    rm edit $(objects)
```

### 24.2.5 添加依赖库

大牛文章[http://blog.csdn.net/qq\\_33850438/article/details/52014399](http://blog.csdn.net/qq_33850438/article/details/52014399)

### 24.2.6 代码泄漏检测

<http://blog.csdn.net/21aspnet/article/details/8172124>

### 24.2.7 参考文献

makefile :<http://blog.csdn.net/liang13664759/article/details/1771246/>  
[http://blog.csdn.net/xzz\\_hust/article/details/8879569](http://blog.csdn.net/xzz_hust/article/details/8879569)



# 第二十五章 调试

## 25.1 Linux - 调试 GDB

### 25.1.1 使用 GDB

**-g 编译项目** 一般来说GDB 主要调试的是C/C++ 的程序。要调试C/C++ 的程序，首先在编译时，我们必须要把调试信息加到可执行文件中。使用编译器（cc/gcc/g++）的 -g 参数可以做到这一点。如：

```
> cc -g hello.c -o hello
> g++ -g hello.cpp -o hello
```

**启动 GDB** 成功编译目标代码以后，让我们来看看如何用gdb 来调试他

`gdb <program>` (program 就是你的执行文件, 不是 .c .cpp .o, 而是加-g 编译后的hello)

### GDB 常用命令

#### 基础命令

- n 单条语句执行，next 命令简写
- r 运行程序，run 命令简写
- c 继续运行程序，continue 命令简写
- p i 打印变量i 的值，print 命令简写
- finish 退出函数
- 设置断点: 我们用break 命令来设置断点。正面有几点设置断点的方法：
  1. break <function> : 在进入指定函数时停住。C++ 中可以使用class::function 或function(type,type) 格式来指定函数名
  2. break <linenum> : 在指定行号停住
  3. break +offset, break -offset: 在当前行号的前面或后面的offset 行停住。offset 为自然数

4. `break ... if <condition>: ...` 可以是上述的参数, `condition` 表示条件, 在条件成立时停住。比如在循环境体中, 可以设置 `break if i=100`, 表示当 `i` 为 100 时停住程序
5. `break *address` 在程序运行的内存地址处停住
6. 查看断点时, 可使用 `info` 命令: `info breakpoints [n]`, `info break [n]` 注: `n` 表示断点号

• **观察点:** 观察点一般来观察某个表达式 (变量也是一种表达式) 的值是否有变化了, 如果有变化, 马上停住程序。我们有下面的几种方法来设置观察点:

1. `watch <expr>`: 为表达式 (变量) `expr` 设置一个观察点。一旦表达式值有变化时, 马上停住程序
2. `rwatch <expr>`: 当表达式 (变量) `expr` 被读时, 停住程序
3. `awatch <expr>`: 当表达式 (变量) 的值被读或被写时, 停住程序
4. `info watchpoints`: 列出当前所设置了的所有观察点

• **捕捉点:** 你可设置捕捉点来捕捉程序运行时的一些事件。如: 载入共享库 (动态链接库) 或是 C++ 的异常。设置捕捉点的格式为:

`catch <event>` 当 `event` 发生时, 停住程序。 `event` 可以是下面的内容:

1. `throw` 一个 C++ 抛出的异常。( `throw` 为关键字)
2. `catch` 一个 C++ 捕捉到的异常。( `catch` 为关键字)
3. `exec` 调用系统调用 `exec` 时。( `exec` 为关键字, 目前此功能只在 HP-UX 下有用)
4. `fork` 调用系统调用 `fork` 时。( `fork` 为关键字, 目前此功能只在 HP-UX 下有用)
5. `vfork` 调用系统调用 `vfork` 时。( `vfork` 为关键字, 目前此功能只在 HP-UX 下有用)
6. `load` 或 `load <libname>` 载入共享库 (动态链接库) 时。( `load` 为关键字, 目前此功能只在 HP-UX 下有用)
7. `unload` 或 `unload <libname>` 卸载共享库 (动态链接库) 时。( `unload` 为关键字, 目前此功能只在 HP-UX 下有用)

`tcatch <event>`

只设置一次捕捉点, 当程序停住以后, 应点被自动删除。

**Help 命令** 启动 `gdb` 后, 就你被带入 `gdb` 的调试环境中, 就可以使用 `gdb` 的命令开始调试程序了, `gdb` 的命令可以使用 `help` 命令来查看, 如下所示

```
/home/hchen> gdb
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show_copyright" to see the conditions.
There is absolutely no warranty for GDB. Type "show_warranty" for details.
This GDB was configured as "i386-suse-Linux".
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

gdb 的命令很多，gdb 把之分成许多个种类。help 命令只是例出gdb 的命令种类，如果要看种类中的命令，可以使用help <class> 命令，如：help breakpoints，查看设置断点的所有命令。也可以直接help <command> 来查看命令的帮助

### 25.1.2 参考

基本使用：<http://www.cnblogs.com/kunhu/p/3603268.html>

大牛教程博客-1:<http://blog.csdn.net/haoel/article/details/2879>

大牛教程博客-2:<http://blog.csdn.net/haoel/article/details/2880>

大牛教程博客-3:<http://blog.csdn.net/haoel/article/details/2881>

大牛教程博客-4:<http://blog.csdn.net/haoel/article/details/2882>

大牛教程博客-5:<http://blog.csdn.net/haoel/article/details/2883>

大牛教程博客-6:<http://blog.csdn.net/haoel/article/details/2884>

大牛教程博客-7:<http://blog.csdn.net/haoel/article/details/2885>



## 第二十六章 工程-细节

### 26.1 语言使用基本问题

#### 26.1.1 注释格式

##### 函数注释

```
/*
 * func()
 *
 * Description of the function.
 *
 * Parameters:
 *   int param1: parameter 1.
 * Returns: int
 *   An integer representing...
 * Throws:
 *   Exception1 if...
 * Notes:
 *   Additional notes...
 */
```

##### 代码注释

```
/*
 * Implements the "insertion sort" algorithm. The algorithm separates the
 * array into two parts--the sorted part and the unsorted part. Each
 * element, starting at position 1, is examined. Everything earlier in the
 * array is in the sorted part, so the algorithm shifts each element over
 * until the correct position is found for the current element. When the
 * algorithm finishes with the last element, the entire array is sorted.
 */
void sort(int inArray[], int inSize)
{
    // Start at position 1 and examine each element.
    for (int i = 1; i < inSize; i++) {
        // Invariant: All elements in the range 0 to i-1 (inclusive) are sorted.
        int element = inArray[i];
```

```

        // j marks the position in the sorted part of the array.
        int j = i - 1;
        // As long as the current slot in the sorted array is higher than
        // the element, shift the slot over and move backwards.
        while (j >= 0 && inArray[j] > element) {
            inArray[j+1] = inArray[j];
            j--;
        }
        // At this point the current position in the sorted array
        // is *not* greater than the element, so this is its new position.
        inArray[j+1] = element;
    }
}

```

## 26.1.2 预编译

单次编译

```

#ifndef MYHEADER_H
#define MYHEADER_H
// ... the contents of this header file
#endif

```

If your compiler supports the `#pragma once` directive, [and](#) most modern compilers [do](#), [this](#) can be rewritten as follows:

```

#pragma once
// ... the contents of this header file

```

条件使用

```

#ifdef [key]
// ...
#endif

```



### 26.1.3 变量问题

变量 (包括成员) 命名

PREFIX	EXAMPLE NAME	LITERAL PREFIX MEANING	USAGE
m m_	mData m_data	"member"	Data member within a class.
s ms ms_	sLookupTable msLookupTable ms_lookupTable	"static"	Static variable or data member.
k	kMaximumLength	"konstant" (German for "constant")	A constant value. Some programmers use all uppercase names to indicate constants.
b is	bCompleted isCompleted	"Boolean"	Designates a Boolean value.
n mNum	nLines mNumLines	"number"	A data member that is also a counter. Since an "n" looks similar to an "m," some programmers instead use mNum as a prefix, as in mNumLines.

使用

- 初始化变量-值化 (int, 指针, 结构体, 堆空间)
- 不定义多余的变量
- 类的构造函数需初始化所有类成员, 否则可能遗留问题
- 引用是某块内存的别名, 不能改, 且不能为空
- 静态成员 (所有对象的值) 需要初始化, 且在类外进行, 初始化时不需要加访问权限, 又因静态成员属于所有对象, 所以初始化应指明其类名。

### 26.1.4 运算符使用问题

•

### 26.1.5 函数问题

•

### 26.1.6 条件语句问题

•

### 26.1.7 循环语句问题

- 

### 26.1.8 数值类型转换问题

- 随机数 <http://blog.csdn.net/beyond0824/article/details/6009908>

## 26.2 内存管理

### 26.2.1 内存分配与使用

- 

### 26.2.2 内存泄漏

- 

## 26.3 缓冲区溢出

### 26.3.1 数组越界

- 

### 26.3.2 数据越界

- 

### 26.3.3 字符串操作溢出

- 

## 26.4 指针问题

### 26.4.1 定义问题

`#define INT_PTR int*` 这是宏定义,编译预处理阶段要进行宏替换,`INT_PTR a,b`会变成 `int* a,b` 所以 `b` 不是指针类型

`typedef int* int_ptr;` 这是自定义类型,也就是把`int_ptr` 定义为 `int` 型指针,编译阶段会把 `c,d` 都识别为指针

### 26.4.2 空指针解引用

- 

### 26.4.3 指针非法使用

- 

### 26.4.4 数组参数

数组做函数参数时，当做普通指针来用

## 26.5 安全缺陷

### 26.5.1 外部输入安全缺陷

- 

### 26.5.2 资源泄漏

- 

### 26.5.3 其他

- 

## 26.6 类

### 26.6.1 类命名问题

- 类名 首字母大写
- 类成员函数 首字母小写，以驼峰样式命名
- 类成员变量 遵照变量命名规则

```
class SpreadsheetCell
{
public:
    void setValue(double inValue);
    double getValue() const;
private:
    double mValue;
};
```

## 26.6.2 访问权限

In C++, a struct can have methods just like a class. In fact, the only difference is that the **default access specifier for a struct is public** while the **default for a class is private**. For example, the SpreadsheetCell class can be rewritten using a struct as follows

```
struct SpreadsheetCell
{
    void setValue(double inValue);
    double getValue() const;
private:
    double mValue;
};
```

## 26.6.3 区分对象所在处

### Object on the Stack

RAII: You create objects **just as you declare simple variables**, except that the variable type is the class name

```
SpreadsheetCell myCell, anotherCell;
myCell.setValue(6);
anotherCell.setString("3.2");
cout << "cell_1:_" << myCell.getValue() << endl;
cout << "cell_2:_" << anotherCell.getValue() << endl;
```

### Object on the Heap

使用 new 操作符 动态的分配的空间一般都处于堆区

```
SpreadsheetCell* myCellp = new SpreadsheetCell();
myCellp->setValue(3.7);
delete myCellp;
myCellp = nullptr;
```

## 26.7 其他

### 26.7.1 预处理

- 

### 26.7.2 异常

-

### 26.7.3 多线程和同步性

- 

### 26.7.4 代码不可达

- 

### 26.7.5 关于 VS2013 中的相对路径问题

- 项目开发的时候，相对路径是以 project.vcproj 为起点
- 分隔符使用 “//”

### 26.7.6 编译

- LNK2001,2005: <http://www.mamicode.com/info-detail-1149828.html>
  - warning C4018: “<”: 有符号/无符号不匹配 [detecot.size() 在容器说明中被定义为: unsigned int 类型, 而 j 是 int 类型所以会出现: 有符号/无符号不匹配警告] [http://blog.csdn.net/huang\\_xw/article/details/8456157](http://blog.csdn.net/huang_xw/article/details/8456157)
  - warning C4316: ... : object allocated on the heap may not be aligned 16  
<http://stackoverflow.com/questions/20104815/warning-c4316-object-allocated-on-the-he>
  - LNK1120: 10 个无法解析的外部命令:<http://bbs.csdn.net/topics/390962489>
  - LNK2019: 无法解析的外部符号 \_\_imp\_\_InitCommonControlsEx@4,该符号在函数 \_WinMainN@16 中被引用
- 项目、属性、链接器、输入、附加依赖项: 填写附加依赖库的名字.lib 空格或分号间隔多项