

## 转换函数

- 将本类型转换为其他类型
- 将其他类型转换为本类型
- 使用 `explicit` 关键字避免隐式转换

## 伪指针(pointer-like classes)和伪函数(function-like classes)

- 伪指针(pointer-like classes)
- 伪函数

## 模板

- 类模板、函数模板和成员模板
- 模板特化和与偏特化
  - 模板特化
  - 模板偏特化
- 模板模板参数

## 引用

- 引用的假象
- 引用的用途:引用被用作美化的指针

## 对象模型

- 成员函数和成员变量在内存中的分布
- 静态绑定和动态绑定

## 常量成员函数

## new 和 delete

- new 和 delete 运算符
- 重载 new 和 delete 运算符
- 重载多个版本 new 和 delete 运算符

# 转换函数

转换函数分为两类: 将本类型转换为其他类型和将其他类型转换为本类型.

## 将本类型转换为其他类型

定义操作符 类型名() 即可指定将本类型变量转换为其他类型的函数,示例如下:

```
1  class Fraction {
2  public:
3      Fraction(int num, int den = 1)
4          : m_numerator(num), m_denominator(den) {}
5
6      operator double() const {    // 重载类型转换运算符 double()
7          return (double)(m_numerator * 1.0 / m_denominator);
8      }
9
10 private:
11     int m_numerator;           // 分子
12     int m_denominator;        // 分母
13 };
```

这种类型转换有可能是隐式的,如下所示:

```

1 Fraction f(3, 5);
2 double d = f + 4;    // 隐式转换,调用 Fraction::operator double() 函数将f转换为
                        double 类型变量

```

对于语句 `f + 4`,编译器可能会去寻找以下重载了运算符 `+` 的两个函数

- `Fraction::operator+(double)`
- `operator+(Fraction, double)`

若这两个函数均没找到,编译器就去寻找能否将 `Fraction` 类型转换为 `double` 类型,找到了类型转换函数 `Fraction::operator double()`,发生了隐式转换.

在上面例子中,若定义了重载运算符 `+` 的函数,就不会再发生隐式转换.

```

1 class Fraction {
2 public:
3     Fraction(int num, int den = 1)
4         : m_numerator(num), m_denominator(den) {}
5
6     explicit operator double() const { // 重载类型转换运算符 double()
7         return (double) (m_numerator * 1.0 / m_denominator);
8     }
9
10    double operator+(double d) const { // 重载运算符 +
11        return (double) (m_numerator * 1.0 / m_denominator) + d;
12    }
13
14 private:
15     int m_numerator;
16     int m_denominator;
17 };

```

```

1 Fraction f(3, 5);
2 double d = f + 4;    // 直接调用 Fraction::operator+(double),不发生类型转换

```

## 将其他类型转换为本类型

类似地,也有可能通过隐式调用构造函数将其他类型的变量转换为本类型,示例如下:

```

1 class Fraction {
2 public:
3     Fraction(int num, int den = 1)
4         : m_numerator(num), m_denominator(den) {}
5
6     Fraction operator+(const Fraction &f) const { // 重载运算符 +
7         return Fraction(m_numerator + f.m_numerator, m_denominator +
8             f.m_denominator);
9     }
10
11 private:
12     int m_numerator;
13     int m_denominator;
14 }

```

```

1 Fraction f1(3, 5);
2 Fraction f2 = f1 + 4;    // 调用 Fraction 类构造函数将 4 转换为 Fraction 类型变量

```

在上面例子中,编译器找不到函数 `Fraction::operator+(int)`,就退而求其次,先隐式调用 `Fraction` 类的构造函数将 4 转换为 `Fraction` 类型变量,再调用 `Fraction::operator+(Fraction)` 函数实现 + 运算.

## 使用 `explicit` 关键字避免隐式转换

使用 `explicit` 关键字可以避免函数被用于隐式类型转换

```

1 class Fraction {
2 public:
3     explicit Fraction(int num, int den = 1)    // 避免隐式调用构造函数进行类型
转换
4         : m_numerator(num), m_denominator(den) {}
5
6     explicit operator double() const {        // 避免隐式调用成员函数进行类型
转换
7         return (double) (m_numerator * 1.0 / m_denominator);
8     }
9
10 private:
11     int m_numerator;
12     int m_denominator;
13 };

```

```

1 Fraction f1(3, 5);
2 Fraction f2 = f1 + 4;    // 编译不通过: error: no match for operator+...
3 double d = f1 + 4;      // 编译不通过: error: no match for operator+...

```

使用 `explicit` 关键字修饰函数后,上述隐式类型转换将不会再发生.

## 伪指针(pointer-like classes)和伪函数(function-like classes)

### 伪指针(pointer-like classes)

伪指针(pointer-like classes)是指作用类似于指针的对象,实现方式是重载 `*` 和 `->` 运算符.

标准库中的 `shared_ptr` 类是一个典型的伪指针类,代码如下:

```

1 template<class T>
2 class shared_ptr {
3 public:
4     T& operator*() const {    // 重载 * 运算符
5         return *px;
6     }
7
8     T *operator->() const {   // 重载 -> 运算符
9         return px;
10    }

```

```

11     //...
12
13 private:
14     T *px;
15     // ...
16 };

```

```

1 int *px = new Foo;
2 shared_ptr<int> sp(px);
3
4 func(*sp);           // 语句1: 被解释为 func(*px)
5 sp -> method();      // 语句2: 被解释为 px -> method()

```

对于语句1,形式上解释得通,重载运算符 `*` 使得 `func(*sp)` 被编译器解释为 `func(*px)`

对于语句2,形式上有瑕疵,重载运算符 `->` 使得 `sp ->` 被编译器解释为 `px`,这样运算符 `->` 就被消耗掉了,只能理解为 `->` 运算符不会被消耗掉.

标准库中的迭代器 `_List_iterator` 也是一个伪指针类,代码如下:

```

1 template<class _Tp, class Ref, class Ptr>
2 struct _List_iterator {
3     _List_iterator& operator++() { ... }
4     _List_iterator operator++(int) { ... }
5     _List_iterator& operator--() { ... }
6     _List_iterator operator--(int) { ... }
7     bool operator==(const _Self &__x) { ... }
8     bool operator!=(const _Self &__x) { ... }
9     Ref operator*() { ... }
10    Ptr operator->() { ... }
11 };

```

`_List_iterator` 除了重载 `*` 和 `->` 运算符之外,还重载了原生指针的其他运算符.

## 伪函数

伪函数(function-like classes)是指作用类似于函数的对象,实现方式是重载 `()` 运算符,标准库中的几个伪函数如下:

```

1 template<class T>
2 struct identity {
3     const T &
4     operator()(const T &x) const { return x; }
5 };
6
7 template<class Pair>
8 struct select1st {
9     const typename Pair::first_type &
10    operator()(const Pair &x) const { return x.first; }
11 };
12
13 template<class Pair>
14 struct select2nd {

```

```

15     const typename Pair::second_type &
16     operator()(const Pair &x) const { return x.second; }
17 };

```

# 模板

## 类模板、函数模板和成员模板

- **类模板**实例化时需要指定具体类型:

```

1  template<typename T>
2  class complex {
3  public:
4      complex(T r = 0, T i = 0)
5          : re(r), im(i)
6      {}
7
8      complex &operator+=(const complex &);
9
10     T real() const { return re; }
11     T imag() const { return im; }
12
13 private:
14     T re, im;
15 }

```

```

1  // 类模板实例化时需要指定具体类型
2  complex<double> c1(2.5, 1.5);
3  complex<int> c2(2, 6);

```

- **函数模板**在调用时编译器会进行参数推导(argument deduction),因此不需要指定具体类型:

```

1  template<class T>
2  inline const T &min(const T &a, const T &b) {
3      return b < a ? b : a;
4  }

```

```

1  // 函数模板实例化时不需要指定具体类型
2  min(3, 2);
3  min(complex(2, 3), complex(1, 5));

```

- **成员模板**用于指定成员函数的参数类型:

```

1  template<class T1, class T2>
2  struct pair {
3      typedef T1 first_type;
4      typedef T1 second_type;
5
6      T1 first;
7      T2 second;
8
9      pair() : first(T1()), second(T2()) {}

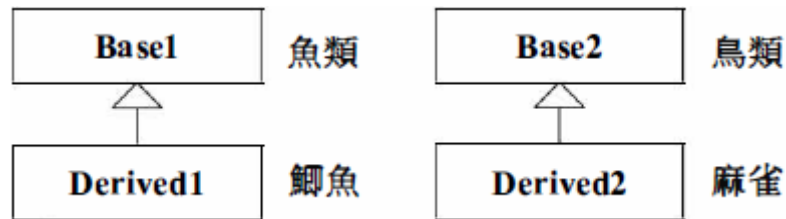
```

```

10     pair(const T1 &a, const T2 &b) : first(a), second(b) {}
11
12     template<class U1, class U2>
13     pair(const pair<U1, U2> &p) : first(p.first), second(p.second) {}
14 }

```

这种结构通常用于实现子类到父类的转换.



```

1 pair<Derived1, Derived2> p1;    // 使用子类构建对象
2 pair<Base1, Base2> p2(p1);    // 将子类对象应用到需要父类的参数上

```

## 模板特化和与偏特化

### 模板特化

模板特化用来部分针对某些特定参数类型执行操作:

```

1  template<class Key>
2  class hash {
3      // ...
4  };
5
6  template<>
7  struct hash<char> {
8      size_t operator()(char x) const { return x; }
9  };
10
11 template<>
12 struct hash<int> {
13     size_t operator()(char x) const { return x; }
14 };
15
16 template<>
17 struct hash<long> {
18     size_t operator()(char x) const { return x; }
19 };

```

上述代码实现针对 `char`、`int` 和 `long` 这三个数据类型使用指定代码创建对象,其它数据类型使用默认的泛化操作创建对象.

### 模板偏特化

模板偏特化有两种形式:

1. 个数的偏: 指定部份参数类型
2. 范围的偏: 缩小参数类型的范围

示例如下:

## 1. 个数的偏:

```
1  template<typename T, typename Alloc>
2  class vector{
3  // ...
4  };
5
6  template<typename Alloc>
7  class vector<bool, Alloc>{ // 指定了第一个参数类型
8  // ...
9  };
```

## 2. 范围的偏

```
1  template<typename T>
2  class C{
3  // 声明1...
4  };
5
6  template<typename T>
7  class C<T*>{ // 指定了参数类型为指针类型
8  // 声明2...
9  };
```

```
1  C<string> obj1; // 执行声明1
2  C<string*> obj2; // 执行声明2
```

# 模板模板参数

模板模板参数是指模板的参数还是模板的情况

```
1  template<typename T, template<typename U> class Container>
2  class XCls {
3  private:
4      Container<T> c;
5  public:
6      // ...
7  };
```

在上面例子里,xcIs 的第二个模板参数 `template<typename U> class Container` 仍然是个模板,因此可以在类声明内使用 `Container<T> c` 语句对模板 `Container` 进行特化,使用方式如下:

```
1  xCls<string, list> mylst1; // mylst1的成员变量c是一个list<string>
```

上面语句构造的 `mylst1` 变量的成员变量 `c` 是一个特化的类 `list<string>`. 仅从模板模板参数的语法来说,上面语句是正确的,但是实际上不能编译通过,因为 `list` 模板有2个模板参数,第二个模板参数通常会被省略,但在类声明体内不能省略其他模板参数,因此可以使用 `using` 语法达到目的:

```
1  template<typename T>
2  using LST = list<T, allocator<T>>
3
4  xCls<string, list> mylst2; // mylst2的成员变量c是一个list<string>
```

这样就能够编译通过了, `mylst2` 的成员变量 `c` 是一个特化的 `list<string>`.

下面这种情况不属于模板模板参数:

```
1  template<class T, class Sequence=deque<T>>
2  class stack {
3      friend bool operator==(const stack &, const stack &);
4      friend bool operator< <>(const stack &, const stack &);
5
6  protected:
7      Sequence c; // 底层容器
8      // ...
9  };
```

上面例子中 `stack` 类的第二模板参数 `class Sequence=deque<T>` 不再是一个模板,而是一个已经特化的类,在实现特化 `stack` 的时候需要同时特化 `class Sequence=deque<T>` 的模板参数.

```
1  stack<int> s1;
2  stack<int, list<int>> s2;    // 特化第二模板参数时应传入特化的类而非模板
```

在上面的例子中 `s2` 在特化时第二模板参数被设为 `list<int>`,是一个特化了的类,而非模板参数,实际上如果愿意的话,甚至可以将第二模板参数设为 `list<double>`,与第一模板参数 `T` 不同,也能编译通过;而模板模板参数就不能这样了,模板模板参数的特化是在类声明体中进行的,类声明体里制定了使用第一模板参数来特化第二模板参数.

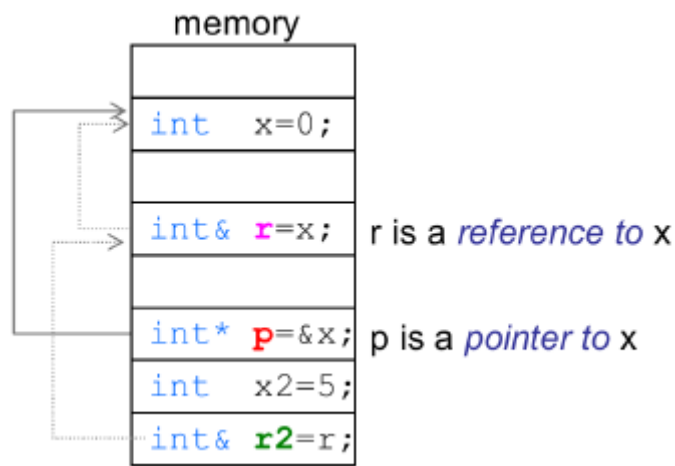
## 引用

声明**引用**(reference)时候必须赋初值,指定其代表某个变量,且之后不能再改变改引用的指向.对引用调用 `=` 运算符同时改变引用和其指向变量的值,不改变引用的指向.

```
1  int x = 0;
2  int *p = &x;
3  int &r = x;    // r代表x,现在r,x都是0
4
5  int x2 = 5;
6  r = x2;        // r不能重新代表其他变量,现在r,x都是5
7
8  int &r2 = r;    // 现在r2,r,x都是5(r2和r都代表x)
```

其内存结构如下所示:





## 引用的假象

虽然在实现上,几乎所有的编译器里引用的底层实现形式都是指针,但C++制造了以下两个**假象**,确保对于使用者来说引用和其指向的变相本身是一致的:

1. 引用对象和被指向的对象的大小相同. (`sizeof(r)==sizeof(x)`)
2. 引用对象和被指向的对象地址相同. (`&x==&r`)

下面程序展示了这两点:

```

1  typedef struct Stag { int a, b, c, d; } s;
2
3  int main(int argc, char **argv) {
4      double x = 0;
5      double *p = &x;    // p指向x, p的值是x的地址
6      double &r = x;      // r代表x, 现在r, x都是0
7
8      cout << sizeof(x) << endl;    // 8
9      cout << sizeof(p) << endl;    // 4, 指针大小为4字节
10     cout << sizeof(r) << endl;    // 8, 假象: r的大小和x相同, 屏蔽了r底层的指针
11
12     cout << p << endl;            // 0065FDFC, x的地址
13     cout << *p << endl;           // 0
14     cout << x << endl;            // 0
15     cout << r << endl;            // 0065FDFC
16     cout << &x << endl;           // 0065FDFC
17     cout << &r << endl;           // 0065FDFC, 假象: r的地址就是x的地址, 屏蔽了r底层的指
    针
18
19     S s;
20     S &rs = s;
21     cout << sizeof(s) << endl;    // 16
22     cout << sizeof(rs) << endl;    // 15
23     cout << &s << endl;            // 0065FDE8
24     cout << &rs << endl;           // 0065FDE8
25
26     return 0;
27 }
```

## 引用的用途:引用被用作美化的指针

在编写程序时,很少将变量类型声明为引用,引用一般用于声明**参数类型**(parameter type)和**返回值类型**(return type).

```
1 // 参数类型声明为引用,不影响函数体内使用变量的方式
2 void func1(Cls obj) { obj.xxx(); } // 值传递参数
3 void func2(Cls *Pobj) { pobj->xxx(); } // 指针传递参数,函数体内使用变量的方式需要修改
4 void func3(Cls &obj) { obj.xxx(); } // 引用传递参数,函数体内使用变量的方式与值传递相同
5
6 // 参数类型声明为引用,不影响参数传递的方式
7 Cls obj;
8 func1(obj); // 值传递参数
9 func2(&obj); // 指针传递参数,传递参数时需要对参数作出修改
10 func3(obj); // 引用传递参数,传递参数时不需对参数做出修改
```

值得注意的是,因为引用传递参数和值传递参数的用法相同,所以两个函数的**函数签名**(signature)相同,不能同时存在.

```
double imag(const double& im) { ... }
double imag(const double im) { ... } // Ambiguity
```

signature

有意思的是,指示**常量成员函数**的 `const` 也是函数签名的一部分,因此 `const` 和 `non-const` 的同名成员函数可以在同一类内共存.

## 对象模型

理解**对象模型**,才能真正理解**多态**和**动态绑定**.

## 成员函数和成员变量在内存中的分布

下面程序在内存中的布局如下所示:

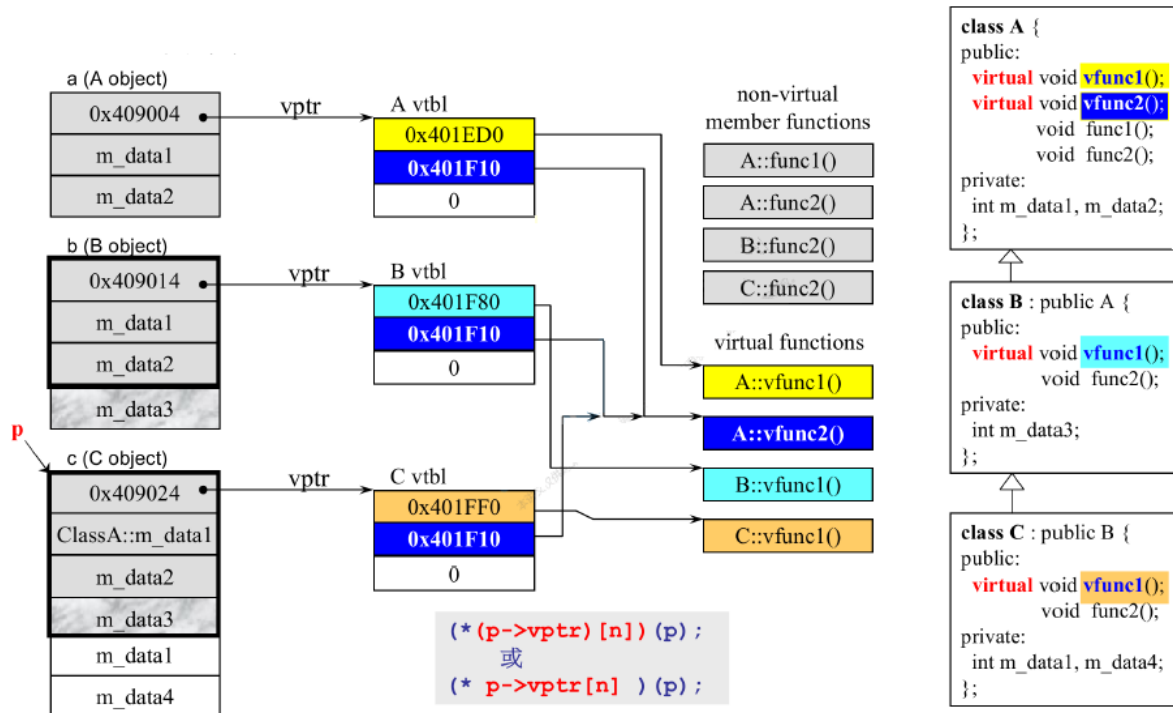
```
1 class A {
2 public:
3     virtual void vfunc1();
4     virtual void vfunc2();
5     void func1();
6     void func2();
7 private:
8     int m_data1;
9     int m_data2;
10 };
11
12 class B : public A {
13 public:
14     virtual void vfunc1();
15     void vfunc2();
16 private:
17     int m_data3;
18 };
```

```

19
20 class C : public B {
21 public:
22     virtual void vfunc1();
23     void vfunc2();
24 private:
25     int m_data1;
26     int m_data4;
27 };

```

其在内存中的布局如下图所示:



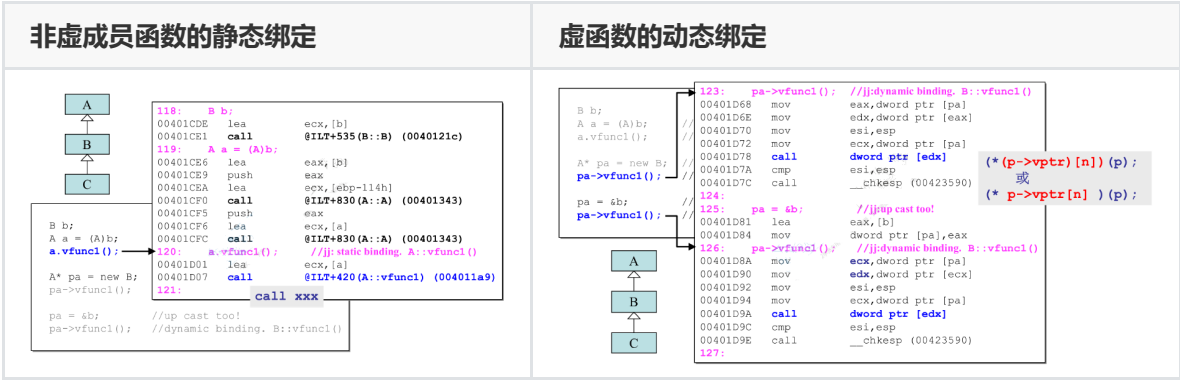
先看成员变量部分: 对于成员变量来说,每个子类对象重都包含父类的成分,值得注意的是, C 类的 m\_data1 字段和父类 A 类的字段 m\_data1 相同,这两个字段共存于 C 类的对象中。

再看函数的部分,每个含有虚函数的对象都包含一个特殊的指针 vptr,指向存储函数指针的虚表 vtbl.编译器根据 vtbl 表中存储的函数指针找到虚函数的具体实现.这种编译函数的方式被称为**动态绑定**.

## 静态绑定和动态绑定

- 对于一般的非虚成员函数来说,其在内存中的地址是固定的,编译时只需将函数调用编译成 call 命令即可,这被称为**静态绑定**.
- 对于虚成员函数,调用时根据虚表 vtbl 判断具体调用的实现函数,相当于先把函数调用翻译成 (\* (p->vptr) [n]) (p),这被称为**动态绑定**.

静态绑定和动态绑定编译出的汇编代码如下所示:



虚函数触发动态绑定的条件是同时满足以下3个条件:

- 1. 必须是通过指针来调用函数.(实测,通过 `.` 运算符调用不会触发动态绑定)
- 2. 指针类型是对象的本身父类.
- 3. 调用的是虚函数.

# 常量成员函数

不改变成员变量的成员函数被称为常量成员函数,在函数体前需要有 `const` 修饰,在[上一节课的笔记](#)中可以看到,若常量成员函数不加以 `const` 修饰,常量对象就无法调用该函数.

是否可以调用	常量对象(const object)	可变对象(non-const object)
常量成员函数(保证不修改成员变量)	✓	✓
非常量成员函数(有可能修改成员变量)	✗	✓

指示常量成员函数的 `const` 被视为函数签名的一部分,也就是说 `const` 和 `non-const` 版本的同名成员函数可以同时存在.

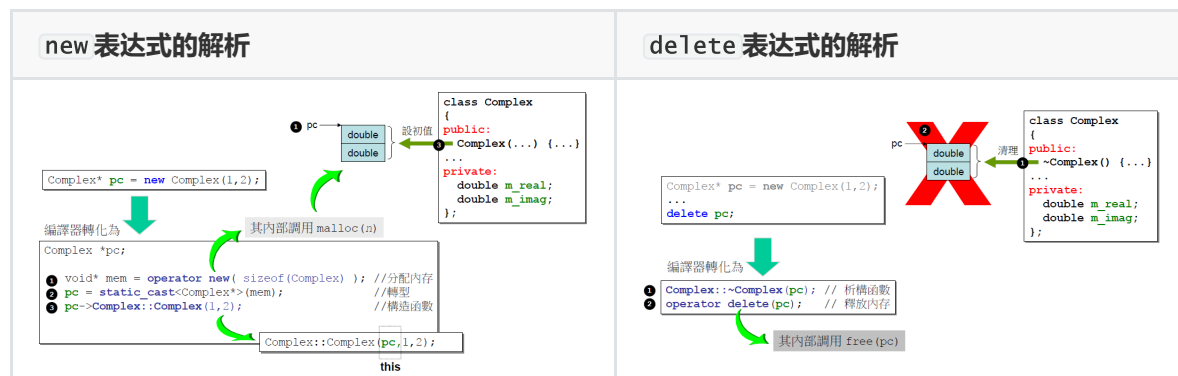
两个版本的同名函数同时存在时,常量对象只能调用 `const` 版本的成员函数,非常量对象只能调用 `non-const` 版本的成员函数.

在STL的 `string` 类重载 `[]` 运算符时,就同时写了 `const` 和 `non-const` 版本的实现函数:

```
1  class template std::basic_string<...> {
2      // ...
3
4      charT operator[] (size_type pos) const {           // 常量成员函数,只有常量对象才能调用该函数
5          // 不用考虑copy on write
6          // ...
7      }
8
9      reference operator[] (size_type pos) const {       // 非常量成员函数,只有非常量对象才能调用该函数
10         // 需要考虑copy on write
11         // ...
12     }
13 }
```

# new和delete

区分 new 表达式和 new 运算符.我们一般程序中写的是 new 表达式,在[上一节课的笔记](#)中可以看到,new 表达式和 delete 表达式会被翻译成多条语句,其中用到了 new 运算符和 delete 运算符.



## new和delete运算符

默认的 new 和 delete 运算符是通过 malloc 和 free 函数实现的,重载这四个函数会产生很大影响,因此一半不应重载这4个函数.

小心，這影響無遠弗屆

```
void* myAlloc(size_t size)
{ return malloc(size); }

void myFree(void* ptr)
{ return free(ptr); }
```

////它們不可以被聲明於一個 namespace 內

```
inline void* operator new(size_t size)
{ cout << "jjhou global new() \n"; return myAlloc( size ); }

inline void* operator new[](size_t size)
{ cout << "jjhou global new[]() \n"; return myAlloc( size ); }

inline void operator delete(void* ptr)
{ cout << "jjhou global delete() \n"; myFree( ptr ); }

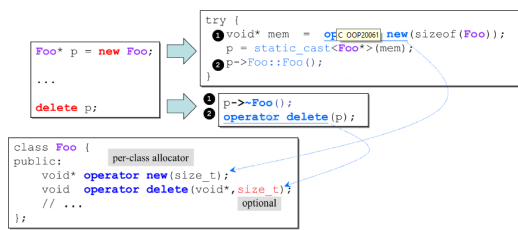
inline void operator delete[](void* ptr)
{ cout << "jjhou global delete[]()\n"; myFree( ptr ); }
```

## 重载 new 和 delete 运算符

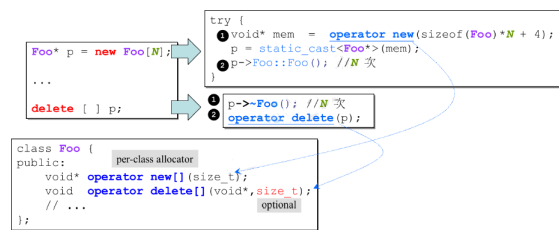
可以在类定义中重载 new、delete、new[] 和 delete[] 运算符,重载之后 new 语句创建该类别实例时会调用重载的 new 运算符.

若重载之后却仍要使用默认的 new 运算符,可以使用 ::new 和 ::delete 语句.

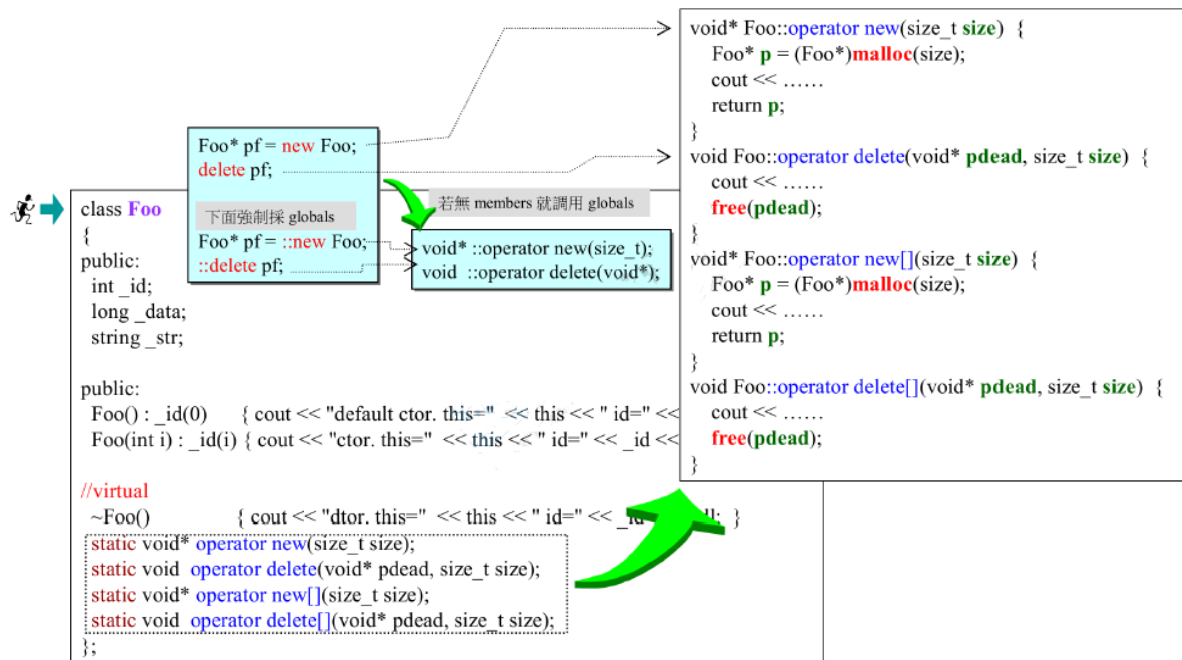
## 重载 new、delete 运算符



## 重载 new[]、delete[] 运算符

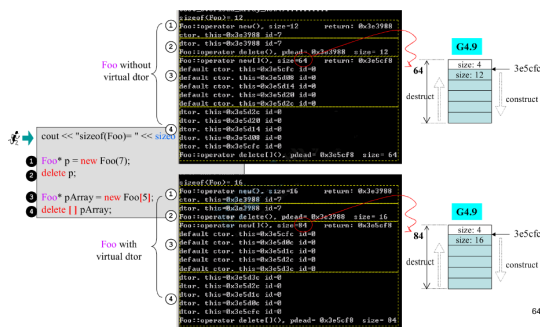


下面例子演示了分别使用重载的 new、delete 运算符和原生 new、delete 运算符的程序运行结果,虫子啊函数的实现如下所示:



程序输出如下所示:

## 调用重载 new、delete、new[]、delete[] 运算符



## 调用默认 new、delete、new[]、delete[] 运算符



从上面程序的执行结果中,可以看出以下几点:

- 含有虚函数的对象不含虚函数的对象的大小多出了4个字节,这4个字节存储虚函数指针 vptr.
- new 运算符接收参数为对象所占字节数, new[] 运算符接收参数为数组所占字节数加4,多出的4个字节用于存储数组长度.
- new[] 和 delete[] 运算符会对数组中每个元素依次调用构造函数和析构函数.

## 重载多个版本 new 和 delete 运算符



- 可以重载多个版本 new 运算符,前提是每个版本都必须有独特的参数列表,且第一个参数必须为 size\_t 类型的,其余参数以 new 语句中指定的参数为初值.

例如类 Foo 重载 new 运算符的函数 operator new(size\_t, int, char) 可以通过语句 Foo \*pf = new(300, 'c') Foo() 调用,第一个括号内的参数为 operator new 的参数,第二个括号内的参数为构造函数的参数.

- 也可以重载多个版本的 delete 运算符,但它们不会被 delete 语句调用.只有当 new 语句所调用的构造函数抛出异常时,才会调用对应的 delete 运算符,主要用来归还未能完全创建成功的对象所占用的内存.

下面例子展示多个重载版本的 new 和 delete 运算符

```

class Foo {
public:
    Foo() { cout << "Foo::Foo()" << endl; }
    Foo(int) { cout << "Foo::Foo(int)" << endl; throw Bad(); }

    // (1) 这个就是一般的 operator new() 的重载
    void* operator new(size_t size) {
        return malloc(size);
    }

    // (2) 这个就是标准库已提供的 placement new() 的重载 (的形式)
    // (所以我也模拟 standard placement new, 就只是传回 pointer)
    void* operator new(size_t size, void* start) {
        return start;
    }

    // (3) 这个才是崭新的 placement new
    void* operator new(size_t size, long extra) {
        return malloc(size+extra);
    }

    // (4) 这又是一个 placement new
    void* operator new(size_t size, long extra, char init) {
        return malloc(size+extra);
    }
}
        
```

class Bad {};

故意在這兒拋出 exception, 測試 placement operator delete.

// (5) 這又是一個 placement new, 但故意寫錯第一參數的 type  
 // (那必須是 size\_t 以符合正常的 operator new)  
 // ! void\* operator new(long extra, char init) {  
 // [Error] 'operator new' takes type 'size\_t' ('unsigned int')  
 // as first parameter [-fpermissive]  
 // ! return malloc(extra);  
 // ! }  
 ...

..... (續上頁)

// 以下是搭配上上述 placement new 的各個所謂 placement delete.

// 當 ctor 發出異常, 這兒對應的 operator (placement) delete 就會被調用.

// 其用途是釋放對應之 placement new 分配所得的 memory.

// (1) 这个就是一般的 operator delete() 的重载

```

void operator delete(void*, size_t)
{ cout << "operator delete(void*, size_t)" << endl; }

// (2) 這是對應上頁的 (2)
void operator delete(void*, void*)
{ cout << "operator delete(void*, void*)" << endl; }

// (3) 這是對應上頁的 (3)
void operator delete(void*, long)
{ cout << "operator delete(void*, long)" << endl; }

// (4) 這是對應上頁的 (4)
void operator delete(void*, long, char)
{ cout << "operator delete(void*, long, char)" << endl; }
        
```

private:

```

int m_i;
};
        
```

即使 operator delete(...) 未能一一對應於 operator new(...), 也不會出現任何報錯. 你的意思是: 放棄處理 ctor 發出的異常.

```

Foo start;
① Foo* p1 = new Foo;
② Foo* p2 = new (&start) Foo;
③ Foo* p3 = new (100) Foo;
④ Foo* p4 = new (100, 'a') Foo;
⑤ Foo* p5 = new (100) Foo(1);
  Foo* p6 = new (100, 'a') Foo(1);
  Foo* p7 = new (&start) Foo(1);
  Foo* p8 = new Foo(1);
        
```

```

Foo::Foo()
① operator new(size_t size), size= 4
Foo::Foo()
② operator new(size_t size, void* start), size= 4 start= 0x22fe8c
Foo::Foo()
③ operator new(size_t size, long extra) 4 100
Foo::Foo()
④ operator new(size_t size, long extra, char init) 4 100 a
Foo::Foo()
⑤ operator new(size_t size, long extra) 4 100
Foo::Foo(int)
terminate called after throwing an instance of 'jj07::Bad'
        
```

奇怪, G4.9 沒調用 operator delete (void\*, long), 但 G2.9 確實調用了.

VC6 warning C4291: 'void \*\_\_cdecl Foo::operator new(~~~)' no matching operator delete found; memory will not be freed if initialization throws an exception [https://blog.csdn.net/ncepu\\_Chen](https://blog.csdn.net/ncepu_Chen)

可以看到,在实际程序运行时,构造函数抛出异常后是否调用对应参数的 delete 运算符与编译器版本有关,是一个比较微妙的事情.

STL 的 string 类重载了 new 操作符以申请额外空间.

