## 构造函数语意学

### 默认构造函数

当程序中未提供任何自定义构造函数时，编译器会在需要的时候生成默认构造函数。需要是指编译器需要，而非程序需要。程序需要时，构造函数应该是程序员负责。

编译器在以下四种情况下需要合成默认构造函数

|  |  |
| --- | --- |
| 成员对象存在默认构造函数 | 成员对象的构造必须调用其默认构造函数，所以合成的默认构造函数中需要调用其默认构造函数。 |
| 基类存在默认构造函数 | 生成派生对象之前先调用构造基类部分，调用基类构造函数。理由同上 |
| 存在虚函数 | 两部分扩张：   1. 产生vtbl 2. 指定vptr |
| 继承链中存在虚基类 | 指定vbptr |

当类中存在自定义构造函数时，以上四种情况编译器会执行代码扩张。

### 复制构造函数

1. 会发生复制行为的三种情况

|  |  |
| --- | --- |
| 显式赋值 | A a**;**  A b **=** a**;** |
| 函数值传参 | void foo**(**A x**){}**  A xx**;**  foo**(**xx**);**  其转化过程为：  1、生成临时变量A \_\_temp**;**  2、\_\_temp.A**::**A**(**xx**);**//拷贝构造函数  3、改变foo的形式为foo**(**A **&**x**);**  4、调用foo**(**\_\_temp**);** |
| 函数返回 | A bar**(){}**  A a **=** bar**();**  转化过程为：  void bar**(**A**&** a**){**  A \_\_temp**;**  **....**  a**.**A**::**A**(**\_\_temp**);**  **}**  将a以参数的形式传递给bar，函数体内初始化一个临时变量\_\_temp，最后使用拷贝构造函数将\_\_temp复制给a |

1. 两种拷贝方式

|  |  |
| --- | --- |
| bitwise copy | 编译器不需要合成拷贝构造函数。浅拷贝 |
| memberwise copy | 深拷贝 |

四种情况下编译器需要合成构造函数，memberwise copy

|  |  |
| --- | --- |
| 成员对象存在拷贝构造函数 |  |
| 基类存在拷贝构造函数 |  |
| 存在虚函数 | class B**{**  public**:**  virtual void f**(){};**  **};**  class D**:**public B**{**  public**:**  virtual void f**(){};**  **};**  D d**;**  B b **=** d**;**//发生截断。如果使用bitwise copy将把d的vptr指针赋值给对象b，错误 |
| 继承链中存在虚基类 | vbptr正确赋值需要依赖拷贝构造函数 |

1. NRV优化和复制构造函数

|  |
| --- |
| 没有NRV优化：  A bar**(){}**  A a **=** bar**();**  转化过程为：  void bar**(**A**&** a**){**  A \_\_temp**;**  **....**  a**.**A**::**A**(**\_\_temp**);**  **}**  将a以参数的形式传递给bar，函数体内初始化一个临时变量\_\_temp，最后使用拷贝构造函数将\_\_temp复制给a  NRV优化：  A bar**(){}**  A a **=** bar**();**  转化过程为：  void bar**(**A**&** a**){**  直接使用a操作，不使用临时变量最后调用拷贝构造函数  **}** |

书中所说的因为没有复制构造函数所以不能做NRV优化，

1. printlocaltime函数

|  |
| --- |
| **<**time**.**h**>**  void plt**(){**  struct tm **\***mtm**;**  time\_t ltime**;**  time**(&**ltime**);**  mtm **=** localtime**(&**ltime**);**  cout**<<**mtm**->**tm\_hour**<<**":"**<<**mtm**->**tm\_min**<<**":"**<<**mtm**->**tm\_sec**<<**endl**;**  **}** |

### 成员初始化列表

1. 四种情况下必须使用初始化列表

|  |  |
| --- | --- |
| const | 不能赋值，只能被初始化 |
| reference | 必须初始化 |
| 类成员没有默认构造函数 | 假设不使用初始化列表时，而使用赋值操作符，赋值操作符之前，该对象成员必须已经被默认初始化，而没有默认构造函数，不能执行默认初始化，假设不成立。 |
| 基类没有默认构造函数 | 理由同上。 |

1. 不同和优点
   1. 不同

|  |  |
| --- | --- |
| 构造函数体内赋值 | 初始化列表 |
| 使用赋值操作符 | 使用构造函数 |
| 会产生临时变量 | 不产生临时变量 |

* 1. 优点

初始化列表由于没有copy操作，效率更高。

1. 顺序
   1. 排序：初始化列表先按照变量声明顺序排序
   2. 扩张：将所有初始化列表代码安插在user code之前。

## Data语义学

假设32位机器：

class X**{};**

class Y**:**virtual public X**{};**

class Z**:**virtual public X**{};**

class A**:**public Y**,**public Z**{};**

**sizeof(**X**);**// 1

**sizeof(**Y**);**// 4 || 8

**sizeof(**Z**);**// 4 || 8

**sizeof(**A**);**// 8 || 12

两种结果分别由于编译器是否对空虚基类做处理得到。

注意两点：

1. 编译器自动添加的数据
2. 对齐

### data member绑定

1. 成员函数中的数据成员绑定，只有在类定义完成之后才会绑定
2. 成员函数中的类型参数，先绑定，遇到新定义再废除先前绑定

### data member布局

只要按照声明顺序出现在内存中，即就是后声明的地址在高位，而不必相邻

### data member存取

对于继承链中没有虚基类的对象，其对象存取和指针存取成本一样。

### 继承和data member

|  |  |
| --- | --- |
| 没有虚函数的单一继承 | 空间膨胀 |
| 有虚函数的单一继承 | 空间时间负担：  空间：vtable  vptr  时间：  constructor内设置vptr  destructor内销毁vtable |
| 多重继承 | 编译期间各个类数据部分的offset都已经计算好，所以成本只有offset计算 |
| 含有虚基类的继承 | 两种实现方式：   |  |  |  | | --- | --- | --- | | \_x |  | 8(表示vptr1的偏移) | | vbptr |  | 12（表共享类偏移） | | vptr1 |  | | \_z |  | | vptr2 |  |  |  |  |  | | --- | --- | --- | | \_x |  | 8(表示共享类offest) | | vptr1 |  | 函数入口地址 | | \_z |  | | vptr2 |  | |

## Function语义学

### 三种函数类型

|  |  |
| --- | --- |
| nonstatic | 转化为nonmember函数   1. 添加对象指针this到参数列表 2. 修改函数体中成员对象，添加this指针 3. mangling处理函数名 |
| static | class A{  static void sf(){};  }；  A bar(){…};  bar().sf();  🡺temp = bar();A::sf(); |
| virtual function | 需要vtable和vptr。  vptr可以放在对象的首部或者尾部。放在首部，可以免去offset计算，c兼容性被破坏；放在尾部更好相反。  具体见下一节 |

### virtual functions

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 单一继承 | |  |  |  | | --- | --- | --- | | vptr1 |  | RTTI，类型 | | \_x |  | 函数入口地址1 | | 派生类data |  | 函数入口地址2 | |
| 多重继承 | |  |  |  | | --- | --- | --- | | vptr1 |  | RTTI | | \_x |  | 函数入口地址 | | vptr2 |  | RTTI | | \_y |  | 函数入口地址 |   多重继承：  对于多重继承下，第二个及其之后的基类，如下调用：  B2 d = new D();  this指针需要thunk，做偏移。以下三种情况需要thunk技术：   1. 第二个及其之后的基类引用派生类，this指针需要做偏移 2. D\*或者B1\*调用B2及其之后的virtual function，this指针需要偏移。 3. 被派生类重写的虚拟函数，返回类型改变。比如：   B2\* d = new D();  d->bar();//返回类型是D\*时，this指针需要偏移到对象的起始地址。 |
| 虚拟继承 | 其虚基类数据成员不像单一继承和多重继承那样出现在对象首部，而是出现在末尾。  class B**{**  public**:**  int a **=** 0**;**  virtual **~**B**(){}**  **};**  class D**:** public virtual B**{**  int b **=** 1**;**  **};**  D**\*** d **=** **new** D**();**  cout**<<**d**<<**endl**;**//  cout**<<(**B**\*)**d**<<**endl**;**  cout**<<sizeof(\***d**)<<**endl**;**  B**\*** db **=** **new** D**();**  cout**<<**db**<<**endl**;**  cout**<<dynamic\_cast<**D**\*>(**db**)<<**endl**;**  cout**<<sizeof(\***db**)<<**endl**;**  输出：  0x7d2238  0x7d2240//加上8字节，指向虚基类成员  16//4（vptr）+4(b)+4(bptr)+4(a)  0x7d2258  0x7d2250//this指针减去8字节  8//4(vptr)+4(a) |

### 指向函数的指针

|  |  |
| --- | --- |
| member function | 指向的是函数地址，参数列表需要对象地址 |
| static function | 只想的内存地址 |
| virtual function | 返回vtable中的索引值 |

### online function

副作用：造成表达式多次求值

临时变量：为了避免表达式多次求值

## 构造、拷贝、析构语义

### 纯虚函数

1. 纯虚函数可以定义和使用，但是只能静态使用
2. 纯虚析构函数必须被定义
3. 不建议将析构函数定义为纯虚函数

|  |
| --- |
| #include <iostream>  **using** **namespace** std**;**  class A**{**  public**:**  virtual void f**()** **=** 0**;**  **};**  void A**::**f**(){**  cout**<<**"f()"**<<**endl**;**  **}**  class D**:**public A**{**  public**:**  void f**(){**  cout**<<**"D f"**<<**endl**;**  A**::**f**();**  **}**  **};**  int main**()**  **{**  D d**;**  d**.**f**();**  **return** 0**;**  **}**  输出：  D f  f() |

### 构造函数

1. 无继承

POD类型编译器并不会自动合成默认构造函数、拷贝、赋值、析构，只会简单的位复制。

1. 有虚函数的无继承

因为编译器会自动在对象的结构中添加vptr，所以简单的位复制会出错：

|  |
| --- |
| #include <iostream>  **using** **namespace** std**;**  class B**{**  public**:**  int x**=**1**;**  **};**  class D**:**public B**{**  public**:**  int y**=**2**;**  virtual void f**(){};**  **};**  int main**()**  **{**  D d**;**  B b **=** d**;**  cout**<<\*((**int**\*)&**d**)<<**endl**;**//4739320  cout**<<\*((**int**\*)&**b**)<<**endl**;**//1  **return** 0**;**  **}**  有输出可见，并非按照位复制，而是&d增加4个字节之后的x值。  这解释了编译器为什么在以下四种情况下需要合成构造、拷贝、赋值和析构：   1. 成员变量有默认构造函数 2. 基类有构造函数 3. 含有虚函数 4. 虚基类 |

1. 存在虚基类时，控制虚基类部分的构造：
2. 构造函数中添加most\_derieved变量，只有当是最深层的类时，才会构造虚基类部分
3. 设置状态，当构造的是subobject时，不构造虚基类部分，只有当构造object时，才会构造虚基类部分。
4. 有虚函数的继承

|  |
| --- |
| 构造函数扩展：   1. 虚基类部分 2. 基类 3. 虚指针 4. initialization list  * 基类为什么要在虚指针设置之前？   c++标准规定：当基类构造函数中调用虚函数时，调用的是自身类的函数。将vprt的设置推迟到基类构造之后，可以保证该规定。   * initialization list为什么在vprt之后？   为了保证initialization list中出现的虚函数能正确应用多态 |