C++ 简介

C++ 从 C 发展而来，除极少数情况外，几乎完全兼容 C，同时又引入了一些新特性。这篇文档是对 C++ 的一个简单介绍，仅供参考。

1. 更安全的类型检查

C 语言允许隐式类型转换，如 short 与 int 类型的变量可以相互赋值，C 编译器会自动完成类型转换，不会有任何提示。

C++ 为了兼容 C，没有禁止隐式转换，但对于不安全的类型转换，C++ 编译器会给出警告信息。如从 short 到 int 的隐式转换，不会丢失数据，C++ 编译器不会说什么。但从 int 到 short 的转换，可能丢失数据，C++ 编译器会无条件发出警告，除非显示的将 int 转换为 short。

对 C++ 编译器的警告，必须足够重视，通常能够避免一些不必要的错误。

2. 内联(inline)函数

在 C/C++ 中，函数调用有一定的开销，频繁调用一些短函数，会导致程序性能下降。C 语言中可以用宏定义避免函数调用，如像下面这样：

```
# define min(x, y)  ((x) < (y) ? (x) : (y))
```

但用宏写的代码，存在一些安全隐患。考虑下面的代码：

```
int x = min(func(), v);
```

按宏定义展开后，func() 函数可能被调用两次，这通常不是我们想要的行为。

C++ 则引入内联函数，兼顾性能的同时，又避免了宏的各种危害。可以显示的将函数声明为 inline，像下面这样：

```
inline int min(int x, int y) {
    return x < y ? x : y;
}
```

编译器，在调用内联函数的地方，直接插入代码，而不是函数调用。需要注意的是，inline 对编译器而言只是一个建议，它不一定会这么做。如对于一个成百上千行的函数，编译器几乎肯定不会内联它。

3. 类与对象

3.1 自定义类型

在 C 语言中，可以用 struct 定义类型，但定义 struct 变量时，必须带上 struct 关键字：

```
struct Apple apple;
```

C 中的 struct 内部只能包含数据，不能定义函数，数据与操作是严格分开的。C++ 中可以用关键字 class 或 struct 定义类型，类中除了数据成员，还可以定义对数据进行操作的函数成员。

C++ 中类的实例称为对象(object)，类与对象的关系，对应于内置类型与变量的关系。定义对象时，不用在类名前面加上 class 或 struct 关键字。按 **Stroustrup** 的说法，他不想让用户定义的类型，成为语言中的二等公民。

下面的代码，基于 pthread 的 pthread_mutex_t 定义了一个 Mutex 类，这种互斥锁在多线程编程中经常用到：

```
class Mutex {
  public:
    Mutex() {
        pthread_mutex_init(&_mutex, NULL);
    }

    ~Mutex() {
        pthread_mutex_destroy(&_mutex);
    }

    void lock() {
        pthread_mutex_lock(&_mutex);
    }

    void unlock() {
        pthread_mutex_unlock(&_mutex);
    }

  private:
    pthread_mutex_t _mutex;
};
```

Mutex 类包含一个数据成员: _mutex，以及几个函数成员，这里暂时只考虑 lock() 与 unlock() 两个函数。类中的函数，必须通过类的对象来调用它们，像下面这样：

```
Mutex mutex;
mutex.lock();
mutex.unlock();
```

以 mutex.lock() 为例，它意味着 lock() 方法被绑定到对象 mutex 上，因此操作的是对象 mutex 中的数据成员 _mutex，而非其他 Mutex 对象中的 _mutex。

方法与对象的绑定是怎么实现的呢？事实上，C++ 内部有一个隐藏的 `this` 指针，指向当前的类对象。上面的 lock() 方法其实有一个隐藏的参数，它的真身如下：

```
void lock(Mutex* this) {
    pthread_mutex_lock(&this->_mutex);
}
```

通过 mutex.lock() 调用 lock() 方法时，C++ 编译器默默的将 mutex 的地址传给了 lock()。

另外指出，`this` 是 C++ 中的一个关键字，可以在类的函数成员中显示使用它，如下所示：

```
void lock() {
    pthread_mutex_lock(&this->_mutex);
}
```

3.2 类的访问修饰符(Access Modifiers)

现在来解释 Mutex 类中出现的 public 与 private，在 C++ 中它们称为类的访问修饰符，同一个修饰符在一个类中可以出现多次。

除 public 与 private 外，还有一个 protected 修饰符。对于访问修饰符，C++ 有如下规定：

- public 成员在类外、类中都可以访问
- private 成员只允许本类中的函数成员访问
- protected 成员在类或子类中可以访问，但不能从类外访问

“访问”的涵义，包括读、写类中的数据成员，以及调用类中的函数。

```
Mutex mutex;
mutex.lock(); // 类外调用 public 函数，没问题
mutex._mutex; // 类外访问 private 成员，编译器报错

// 再看看 Mutex 中 lock() 的实现
void lock() {
    pthread_mutex_lock(&_mutex); // lock() 是类中函数成员，访问 private 成员 _mutex，没问题
}
```

3.3 构造函数与析构函数

接下来解释 Mutex 类中出现的 Mutex() 与 ~Mutex()，它们分别被称为类的构造函数、析构函数。有下面几点需要注意：

- 构造函数、析构函数没有返回值，不能在函数名前面加 void，不信你试试
- 构造函数名字与类名相同，可以带参数
- 析构函数名字在类名前加了一个 ~，不能带参数
- 创建类对象时，C++ 自动调用类的构造函数
- 销毁类对象时，C++ 自动调用类的析构函数

以 `Mutex mutex;` 为例，C++ 会创建一个 Mutex 对象，并调用 Mutex 的构造函数，进行初始化工作。而当 mutex 超出生存期时，C++ 会调用 Mutex 的析构函数，释放系统资源。

在 C 中定义一个 struct 变量时，必须手动调用初始化代码，而它寿终正寝时，还得手动处理它的后事。在 C++ 中，这些杂活可以一次性写入构造函数与析构函数，然后交给 C++ 编译器就好了，它会在需要的地方插入相关的代码。与 C 相比，C++ 就像是一个优雅的仆人。

还有一点要注意，不能通过 `mutex.Mutex()` 手动调用构造函数，原因在于构造函数在内部是匿名(没有名字)的，Mutex 类中根本就不存在名字为 Mutex 的函数。然而，C++ 允许调用析构函数，像下面这样：

```
mutex.~Mutex();
```

建议不要这么做，它会导致析构函数被调用两次，程序很可能因此崩溃。

3.4 额外的函数调用开销？

可能有人担心 C++ 的构造函数、析构函数与 C 相比，多了一次函数调用的开销。然而，不要忘了内联。C++ 规定在类的内部定义的函数，默认是内联的。因此，上述 Mutex 类中的所有函数，包括构造函数、析构函数，都是内联的，经过编译器优化后，不会有任何额外的函数调用开销。

类的函数成员也可以在类外定义，像下面这样：

```
// 类中仅声明函数: void lock();
inline void Mutex::lock() {
    pthread_mutex_lock(&_mutex);
}
```

需要注意的是，得加上 `Mutex::` 限定符，以表明 lock() 是 Mutex 类中的成员函数。另外，如果希望编译器内联它，必须显示声明为 inline。

3.5 动态创建对象

在 C 中，可以像下面这样动态的创建或释放一个对象：

```
struct T* t = (struct T*) malloc(sizeof(*t));
init(t);
destroy(t);
free(t);
```

C++ 中不能这样做，malloc 与 free 只负责分配、释放内存，不会调用构造函数、析构函数。C++ 中分别用 `new` 与 `delete` 取代 `malloc` 与 `free`。上面那段代码的 C++ 版本看起来像这样：

```
T* t = new T; // 分配内存，并调用 T 的构造函数
delete t;     // 调用 T 的析构函数，并释放内存
```

如果调用的是带参数的构造函数，代码可以写成这样：

```
T* t = new T(para1, para2);
```

特别的，对内置类型也可以进行 new 与 delete 操作：

```
int* p = new int(3); // ok. *p = 3
delete p;
```

另外，如果需要动态创建对象的数组，可以用 new[] 与 delete[] 操作符，代码如下：

```
T* t = new T[3];
delete[] t;
```

但是有一个限制，没办法调用类的带参数的构造函数，如下的代码是错误的：

```
T* t = new T[3](2);
T* t = new T(2)[3];
```

C++11 中提供了一个解决办法：

```
T* t = new T[3] {2, 2, 2};
```

但是，当创建对象的数量比较多时，仍然会有麻烦。无论如何，new[] 与 delete[] 不是什么优雅的东西，应该尽量避免在代码里使用它们。

4. 函数重载

C++ 允许函数重载(function overloading)，即定义多个同名的函数。重载这个词可以理解为，给同一个名字定义不同的意义，像文言文中的一词多义一样。而名字具体表达的意义，取决于它的使用场景(传的什么参数)。

引入函数重载的原因之一是，定义多个版本的构造函数，通常会有好处。如下述代码：

```
std::string s("hello world"); // ok. s = "hello world"
std::string s(5, 'a');        // ok. s = "aaaaa"
```

编译器会根据传递的参数类型，决定调用哪个同名函数。如果存在参数类型完全匹配的函数，当然最好，否则编译器会根据隐式类型转换的规则，选择匹配度最高的那个函数。

需要注意的是，函数重载只与函数的参数有关，而与函数的返回值无关。因为在一般情况下，编译器不能通过函数的返回值判断你要调用哪个版本的函数。像下面这样的代码，虽然看起来比较合理，但编译器会报错：

```
void fun() {}
int fun() { return 0; }
int x = fun();
```

调用一个函数，需要知道它的地址。在 C 中函数名是全局唯一的，因此可以用函数名标志函数地址。在 C++ 中，引入函数重载后，函数名的全局唯一性被破坏了，不可能用同一个名字，标志多个重载函数的地址。

为了解决这个问题，C++ 采用了 **name mangling** 机制，即重载的同名函数，在内部实现中有着不同的名字。但 C++ 标准对函数内部的名字并没有任何规定，不同的编译器可能使用了不同的实现方法。这样就不能方便的得知函数在内部的真实名字，当然也就不能轻松的从内部名字还原(demangling)出函数的本来面目。

5. 操作符重载

操作符本质上相当于函数，因此操作符重载也就是函数重载。下面以一段简短的代码进行说明：

```
Apple operator+(Apple a, Apple b) {
    // do something here
}

Apple a, b, c;
c = a + b;           // ok. 等价于 c = operator+(a, b)
c = operator+(a, b); // ok. 直接调用 operator+ 函数
```

上面的代码为 Apple 类定义了加法操作。可以看到，重载 `+` 只需要定义一个名为 `operator+` 的函数，`+` 前面可以加空格。`operator+` 这个名字确实有些奇怪，但它事实上就是一个函数而已，可以直接调用它。

其他操作符的重载是类似的，定义相应的函数就行了。因为函数可以在类中定义，所以也可以在类中重载操作符，只是要少传一个参数，因为编译器默认会传 `this` 指针，操作符的第一个参数就是 `this` 所指的当前对象。

C++ 中允许重载的操作符有几十种，但常用的并不多，不用浪费太多时间在上面，必要时可以参考 **cppreference**。

6. 引用

6.1 为什么需要引用？

引入操作符重载后，就有一个新问题，不能像普通函数那样传递指针：

```
Apple operator+(Apple* a, Apple* b); // 编译器报错，不允许传指针
Apple a, b;
&a + &b; // 如果允许传指针的话，就得写出这种奇怪的东西
```

因为操作符重载不能传指针，为了解决大对象的传参问题，就需要引入引用的概念。

6.2 引用的基本概念

在 C++ 中，类型名后面加上符号 & 就表示这个类型的引用。

```
int x = 3;
int& y = x; // y is a reference to x
y = 7;      // ok. now x = 7
```

上述代码中，y 是变量 x 的引用，通过操作 y 最终改变了 x 的值。可以看出，与变量、指针相比，引用是截然不同的概念。

6.3 深入理解引用

- 变量的本质

考虑上面代码中的 `int x = 3;`，编译器会申请一块 4 个字节的内存，并将内存的值设成 3，然后将变量名 x 绑定到这块内存。

注意，“绑定”的涵义是指：代码中对 x 的操作，就是对 x 所代表的内存的操作(读、写内存中的值)。如在代码中执行 `x = 7`，实际上就是将 x 代表的内存的值设成 7。

因此，变量本质上就是一块内存，变量名被绑定到这块内存上，通过变量名可以操作这块内存。

- 指针的本质

考虑代码 `int* p = &x;`，在 64 位机器上，编译器会申请一块 8 个字节的内存，并将内存的值设成变量 x 的地址，然后将名字 p 绑定到这 8 个字节的内存上。注意这里的 `int*`，它也是一种类型。

因此，指针也是变量，本质上也是一块内存，只不过内存中存储的是某个变量的地址。

- 引用的本质

现在再来看引用。考虑上面的代码 `int& y = x;`，编译器在这里并没有分配内存，它只是将名字 y 也绑定到变量 x 所表示的内存上，所以通过名字 y 也可以操作 x 所表示的内存。

定义一种类型的变量(对象)时，编译器会分配一块内存，而定义引用时并没有内存分配的操作。因此，引用并不是一种类型，“引用类型”这种说法是错误的。本质上，引用只是给某块内存绑定了一个新的名字。

6.4 引用的注意事项

在 C++ 中定义引用时，必须初始化，而且初始化后，没办法更改，不可能再让它引用别的变量(对象)。像下面的代码都是错误的：

```
int& a;           // 没有初始化
int& b = 7;       // 引用必须绑定一块内存，7 并不表示一块内存
const int x = 3;
int& y = x;       // 通过 y 可以修改 x 在内存中的值，但 x 是常量，编译器报错

int i = 3;
double& d = i;    // double& 不能引用 int 类型的变量
```

6.5 const 引用

引用可以用 const 修饰，const 引用对外宣称它引用的是 const 对象(实际上可能并不是 const 的)，不能通过它改变这个对象。以下面的代码为例：

```
int a = 3;
const int b = 3;
const int& x = a;  // ok. x 是 a 的一个引用，但不能通过 x 修改 a
const int& y = b;  // ok.
const int& z = 7;  // ok. 编译器生成一个临时变量保存 7 这个值，z 引用的是临时变量
x = 7;             // ko. 不能通过 const 引用修改变量的值
```

const 引用可以引用变量、常量，甚至像 7 这样的字面常量，只是不能通过它修改所引用的变量或对象。

注意上面的 `const int& z = 7;`，编译器在这里生成一个临时变量保存 7，因为对临时变量的 const 引用是无害的。而对于 `int& x = 7;`，编译器会直接报错，而不会生成一个临时变量供 x 引用，因为改变一个临时变量通常不是我们想要的行为。

6.6 在函数参数中传递引用

前面已经说过，引用本质上就是给一块内存绑定一个新的名字。在函数参数中传递引用，编译器会将形参的名字，直接绑定到所引用对象的那块内存上，不存在内存拷贝，也不会传递地址。详情见下面的代码：

```
void f(const int& v) {
    // do something here
}

void g(int& v) {
    // do something here
}

int x = 3;
const int y = 3;

f(3); // ok. 参数名 v 被绑定到临时变量所对应的那块内存
f(x); // ok. 参数名 v 被绑定到变量 x 所对应的那块内存
f(y); // ok. 参数名 v 被绑定到常量 y 所对应的那块内存

g(3); // ko. 非 const 引用不能引用字面常量
g(x); // ok. 参数名 v 被绑定到变量 x 所对应的那块内存
g(y); // ko. 非 const 引用不能引用常量
```

7. 类的初始化列表

如果类中含有引用、常量，怎么初始化它们呢？不能在构造函数中初始化它们，所以得引入初始化列表的概念。下面的代码给出了一个例子：

```
class MutexGuard {
  public:
    explicit MutexGuard(Mutex& mutex)
        : _mutex(mutex) {
        _mutex.lock();
    }

    ~MutexGuard() {
        _mutex.unlock();
    }

  private:
    Mutex& _mutex; // 类中的引用，C++ 内部可能会用指针实现
};
```

注意代码中的 `: _mutex(mutex)`，这个 `:` 开头的区域就称为初始化列表，如果有多项，可以用逗号隔开。这个区域的初始化工作在进入构造函数之前完成。

另外，如果类中包含其他类的对象，那么这些对象的初始化一般也要放到初始化列表中，因为没有办法显示调用类的构造函数。

如果这些类定义了默认构造函数，即不带任何参数的构造函数，那么可以不将它们放到初始化列表中，因为编译器会自动调用默认构造函数进行初始化。

关于初始化列表的更多详情，可以参考这里。

8. 拷贝构造函数与赋值操作

8.1 定义拷贝构造函数与赋值操作

直接以一段代码进行说明：

```
class Apple {
  public:
    Apple() {
        _v = 0;
    } // 默认构造函数

    Apple(int v) {
        _v = v;
    } // 带参数的构造函数

    Apple(const Apple& another) {
        if (this == &another) return; // 防止自我拷贝
        _v = another._v;
    } // 拷贝构造函数

    Apple& operator=(const Apple& another) {
        if (this == &another) return *this; // 防止自我赋值
        _v = another._v;
```

```
        return *this;
    } // 赋值操作

  private:
    int _v;
};

Apple x;      // ok. 调用默认构造函数进行初始化
Apple y(3);   // ok. 调用带参数的构造函数进行初始化
Apple z = x; // ok. 调用拷贝构造函数进行初始化
z = y;        // ok. 调用赋值操作
```

注意执行 `Apple z = x;` 时，z 还没有初始化，因此调用的是拷贝构造函数。而执行 `z = y` 时，z 已经初始化了，而一个对象只能被初始化一次，因此这里调用的是赋值操作。

8.2 禁用拷贝构造函数与赋值操作

在大多数时候，我们不希望类支持拷贝或赋值操作。为了不让编译器自动生成这些函数，可以使用下面的小技巧，定义一个宏：

```
#define DISALLOW_COPY_AND_ASSIGN(Type) \
    Type(const Type&); \
    void operator=(const Type&)
```

要禁用拷贝、赋值操作，以 Apple 类为例，只需要在 Apple 类的 private 域加上：

```
DISALLOW_COPY_AND_ASSIGN(Apple);
```

这里将拷贝构造函数与赋值操作声明为私有的，并且不定义它们。这样的话，代码中试图对该类进行拷贝、赋值操作时，编译器就会报错。

指出一点，上述 `operator=` 的返回值是 void，因为函数重载与返回值无关。其实，返回一个 `Type&` 也是可以的。

在 `C++11` 中，也可以将上面的宏写成如下的形式，好处是不用放到 private 域中：

```
#define DISALLOW_COPY_AND_ASSIGN(Type) \
    Type(const Type&) = delete; \
    void operator=(const Type&) = delete
```

9. 模板

模板是个强大而且非常有用的特性。C++ 提供了一套标准模板库 **STL**，里面用到了各种模板技术。不过，这里只能简单的介绍下模板的概念。

9.1 函数模板

```
template <typename T>
inline T add(T x, T y) {
    return x + y;
}
```

上面定义了一个函数模板，调用 `add(1, 1)` 时，编译器会生成实际的代码：

```
inline int add(int x, int y) {
    return x + y;
}
```

同理，调用 `add(1.0, 2.3)` 时，编译器则会生成 `double` 版的代码。

9.2 类模板

```
template <typename T>
class unique_ptr {
  public:
    unique_ptr(T* ptr = 0) {  // 提供默认参数 0
        _ptr = ptr;
    }

    ~unique_ptr() {
        delete _ptr; // 释放内存，delete NULL 不会有问题
    }

    T* get() const { // const 成员函数表示不修改类中的数据，仅此而已
        return _ptr;
    }

    // 重载操作符 -> 与 *，使得 unique_ptr 的行为更像指针
    T* operator->() const {
        //assert(_ptr != 0);
        return _ptr;
    }

    T& operator*() const {
        //assert(_ptr != 0);
        return *_ptr;
    }

    void reset(T* ptr = 0) {
        delete _ptr;
        ptr = _ptr;
    }

    T* release() {  // 交出指针所有权
        T* ptr = _ptr;
        _ptr = 0;
        return ptr;
    }

  private:
    T* _ptr;
    //DISALLOW_COPY_AND_ASSIGN(unique_ptr);
};
```

上面的代码，实际上是 `C++11` 中 `std::unique_ptr` 的一个简单实现，用到了模板、操作符重载。`unique_ptr` 类的用法很简单：

```
unique_ptr<int> pi(new int(7));
std::cout << "*pi = " << *pi << std::endl; // *pi 等价于 pi.operator*()

unique_ptr<Apple> pa(new Apple);
pa->turn_red();              // 假设 Apple 类有一个 turn_red 方法
pa.operator->()->turn_red(); // 等价于上面的调用方式
```

10. 命名空间(namespace)

在 C 中，函数、全局变量都在全局空间，所以取名字时，必须非常小心，以免出现冲突。C++ 则引入命名空间，不同项目或模块的名字，可以放到不同的命名空间。C++ 标准库提供的东西，都在命名空间 `std` 中。

```
// 在 namespace str 中声明 split 函数
namespace str {
std::vector<std::string> split(const std::string& s, char sep);
}

str::split("hello world", ' '); // 调用 split 函数

namespace xx = ::std; // 给 std 取别名，:: 表示全局空间
using namespace std;  // 将 std 中的名字全部引入当前命名空间，不推荐

using std::cout;
using std::endl;
cout << "hello world" << endl;
```

11. 继承

C++ 中的继承机制比较复杂，既支持单继承，又支持多继承。另外，继承又分为 public、protected、private 继承。呃.. 还有 virtual 继承..

```
class X : public A, protected B, private C, public virtual D {
    // bla bla bla......
};
```

上面是个极端的例子，类 X 有 4 个基类，分别采用了不同的继承方式。A，B，C 分别被称为 X 的 public，protected，private 基类，而 D 则是 X 的 public 虚基类。虚基类中的 virtual 与虚函数中的 virtual 是两个完全不同的概念。

public、protected、private 继承的区别在于，基类成员在子类中的访问权限不同：

- public 继承最常用，基类成员的访问权限在子类中保持不变。
- protected 继承很少用，基类的 public 成员在子类中会变成 protected 成员。
- private 继承偶尔会用到，基类的所有成员在子类中都变成 private 成员。

而引入 **virtual 继承**，则是为了解决多继承中的棱形继承问题(有些 C++ 面试官会问到)。实际代码中很少遇到这种问题。

用继承写出的代码，可读性比较差，不容易理解。因此，在实际项目中建议：

- 尽量用组合取代继承。
- 尽量避免使用多继承。
- 尽量避免三层及以上的继承关系。

```
class A {
};

// 继承
class B : public A {
};

// 组合
class B {
    A _a;
};

// 组合比继承更灵活，可以只保存对象的指针，以减轻代码的耦合性
class B {
    A* _a; // 类 A 发生变化时，不用重新编译类 B 的代码
};
```

12. 虚函数

12.1 为什么引入虚函数？

C++ 引入虚函数(virtual function)的目的很简单，就是为了解决用基类指针或引用调用子类方法的问题。

考虑下面的代码：

```
#include <iostream>

class A {
  public:
    void fun() {
        std::cout << "A::fun() is called" << std::endl;
    }
};

class B : public A {
  public:
    void fun() {
        std::cout << "B::fun() is called" << std::endl;
    }
};

int main() {
    B b;
    A* pa = &b;
    A& ra = b;
    pa->fun();
    ra.fun();
    return 0;
}
```

上述代码编译后执行结果如下：

```
A::fun() is called
A::fun() is called
```

没有虚函数的机制，上述 `fun()` 始终被静态绑定(编译期绑定)到基类 A 中的版本，而不会调用子类 B 中的 `fun()`。

现在加上虚函数：

```
#include <iostream>

class A {
  public:
    void fun() {
        std::cout << "A::fun() is called" << std::endl;
    }

    virtual void vfun() {
        std::cout << "A::vfun() is called" << std::endl;
    }
};

class B : public A {
  public:
    virtual void fun() {
        std::cout << "B::fun() is called" << std::endl;
    }

    virtual void vfun() {
        std::cout << "B::vfun() is called" << std::endl;
    }
};

int main() {
    B b;
    A* pa = &b;
    A& ra = b;
    pa->fun();
    ra.fun();
    pa->vfun();
    ra.vfun();
    return 0;
}
```

编译后执行结果如下：

```
A::fun() is called
A::fun() is called
B::vfun() is called
B::vfun() is called
```

可以看到，`fun()` 在基类 A 中不是虚函数，通过 A 类指针或引用调用的始终是 A 类中的版本。尽管 `fun()` 在子类 B 中被声明为 virtual，但没有造成任何影响。

而 A 中的虚函数 `vfun()` 则执行动态绑定，A 类指针(引用)实际指向(引用)的是 B 类对象，所以最终调用的是 B 中的 `vfun()`。

12.2 虚析构函数

带有虚函数的类，析构函数一般也要声明为 virtual。考虑下面的代码：

```
#include <iostream>

class A {
  public:
    A() {
        std::cout << "A() is called" << std::endl;
    }

    ~A() {
        std::cout << "~A() is called" << std::endl;
    }
};

class B : public A {
  public:
    B() {
        std::cout << "B() is called" << std::endl;
    }

    ~B() {
        std::cout << "~B() is called" << std::endl;
    }
};

int main() {
    A* pa = new B;
    delete pa;
    return 0;
}
```

上面的代码，执行结果如下：

```
A() is called
B() is called
~A() is called
```

可以看到，尽管 `pa` 实际指向的是 B 类对象，`delete pa` 时调用的却是 A 的析构函数。因为 A 的析构函数不是 virtual 的，编译器执行的是静态绑定。

将 A，B 中的析构函数声明为 virtual：

```
#include <iostream>

class A {
  public:
    A() {
        std::cout << "A() is called" << std::endl;
    }

    virtual ~A() {
        std::cout << "~A() is called" << std::endl;
    }
};

class B : public A {
  public:
    B() {
        std::cout << "B() is called" << std::endl;
```

```
    }

    virtual ~B() {
        std::cout << "~B() is called" << std::endl;
    }
};

int main() {
    A* pa = new B;
    delete pa;
    return 0;
}
```

重新编译执行后，结果如下：

```
A() is called
B() is called
~B() is called
~A() is called
```

现在 `delete pa` 调用的就是 B 的析构函数，不要误认为编译器先调用 B 的析构函数，再调用 A 的析构函数，A 的析构函数被调用，是因为编译器在 B 的析构函数中插入了调用 A 类析构函数的代码。

总之，类中含有虚函数时，其析构函数一般也必须是 virtual 的，不然 delete 指向子类对象的基类指针时，子类的析构函数不会被调用。

12.3 虚函数的内部实现机制

C++ 内部实现中，每个包含虚函数的类，都有一张虚函数表(virtual table)，保存类中可调用的虚函数的地址。编译器会在类中，插入一个指针(vptr)，指向类的虚函数表。

考虑下面的代码：

```
class A {
  public:
    virtual void f1() {}
    virtual void f2() {}
};

class B : public A {
  public:
    virtual void f1() {}
};

class C : public A {
  public:
    virtual void f2() {}
};
```

A，B，C 三个类的 vptr 与 vtbl 分别如下：

```
A::_vptr  ->       vtbl_A
               --------------
                    vp1    ->    A::f1
                    vp2    ->    A::f2
```

```
B::_vptr  ->       vtbl_B
                --------------
                     vp1    ->    B::f1    // 覆盖 A::f1
                     vp2    ->    A::f2    // 继承 A::f2

C::_vptr  ->       vtbl_C
                --------------
                     vp1    ->    A::f1    // 继承 A::f1
                     vp2    ->    C::f2    // 覆盖 A::f2
```

A，B，C 中都只有一个 vptr，构建对象时，vptr 会被初始化，指向实际类型的 vtbl。下面的代码展示了 vptr 是如何工作的：

```
B b;         // _vptr -> vtbl_B
A* pa = &b;
pa->f1();    // (*pa->_vptr[0])()   实际调用的是 vtbl_B 中 vp1 指向的 B::f1
pa->f2();    // (*pa->_vptr[1])()   实际调用的是 vtbl_B 中 vp2 指向的 A::f2
```

上面只涉及到单继承，在多继承中，情况会复杂些，类中可能不只一个 vptr，详情可以参考这里。

12.4 纯虚函数

C++ 中引入了纯虚函数(pure virtual function)的概念，声明纯虚函数的语法比较丑陋：

```
class X {
  public:
    virtual void xx() = 0;  // 函数声明后面加上 = 0，表示纯虚函数
};
```

含有纯虚函数的类，被称为抽象类，不允许实例化，即不能创建抽象类的对象。不能创建对象的类，有什么用？对外提供统一的抽象接口，而由子类实现这些接口。

```
class AbstractFs {
  public:
    virtual ~AbstractFs() {}

    // 为了简单，略去了函数的返回值及参数
    virtual void create() = 0;
    virtual void open() = 0;
    virtual void read() = 0;
    virtual void write() = 0;
    virtual void close() = 0;
    // ......
};

// 实现一种文件系统
class Ext4Fs : public AbstractFs {
  public:
    // 实现文件系统的操作......
};

// 给一种文件系统添加输入、输出日志
class DebugFs : public AbstractFs {
  public:
    virtual void create() {
        // 打印 create 的输入参数
```

```
        _fs->create();
        // 打印 create 的输出结果
    }

    // ......

  private:
    AbstractFs* _fs;  // _fs 指向已经实现的一种文件系统
};
```

13. 标准库

C++ 标准库非常精简，学习成本相对比较低。尤其是 C++ 的标准模板库 **STL**，实现了常见的数据结构，用起来非常方便。可以去 **cplusplus** 上查看 STL 的用法。

14. 推荐的 C++ 参考书

- 入门
  很多人推荐 **C++ Primer**，有点厚，可以跳过无关紧要的内容。
- 进阶
  Scott Meyers 的 **Effective C++** 不错，可以吸收一些有用的经验。
- 深入
  Stanley B. Lippman 的《深度探索 C++ 对象模型》(侯捷译) 值得一看，可以深入了解 C++ 的内部实现细节。
- 语言设计
  Bjarne Stroustrup 的《C++语言的设计和演化》，可以从语言设计层面了解 C++ 引入的新特性。