指针与内存

class与内存管理

上节课我们介绍了C++面向对象编程的思想，还有运算符重载、友元函数，以及一些小的注意事项。

但是，大家可能已经发现了，上节课所讲述的例子中不涉及动态内存分配的操作，所有变量均为自动变量，其所占用的内存空间会在变量的生命周期结束之后自动释放。而动态分配的内存不会如此，我们需要手动地用 `delete pointer` 的语句来释放它的内存空间，同时，当我们在面向对象编程时运用动态内存分配时，也会带来一些问题，需要额外的注意，这里为大家介绍常见的一些要点。

仍然以一个例子开始，实现一个简单的字符串类，支持如下操作

```
int main(){
  String s1();
  String s2("hello");
  String s3(s1);    // 调用拷贝构造函数
  s3 = s2;          // 调用拷贝赋值函数

  cout<<s3<<endl;
}
```

String class的声明

```
class String
{
public:
  String(const char * cstr = nullptr);
  String(const String& str);                // 拷贝构造 Big Three
  String & operator = (const String& str);  // 拷贝赋值 Big Three
  ~String();                                // 析构函数 Big Three
  char * get_c_str() const {return m_data}; // 获取C风格字符串

private:
  char *m_data;
};
```

自定义的拷贝构造函数和拷贝赋值函数

如果没有编写拷贝构造、拷贝赋值两个函数，编译器会为我们提供默认的一套，其功能是将对象的字节原封不动地复制过去。

但是，如果class带有指针成员，并且在运行时会进行动态内存分配，则使用默认的拷贝构造和拷贝赋值会造成新对象的指针重复指向同一段内存，这并不是真正的“新对象”，会带来意想不到的错误，故这时需要自定义这两个函数，以使得它们的行为满足预期。

同理，析构函数也是如此，编译器默认给出的析构函数只会释放class成员所占用的空间，而对于`m_data`所指向的空间，默认的析构函数就无能为力了。

拷贝构造、拷贝赋值、析构这三个函数被称为 *Big Three*，当我们设计的class里面包含指针成员时，需要特别关注这三个函数。

构造&析构

```
inline
String::String(const char * cstr){
  if(cstr){
    m_data = new char[strlen(cstr)+1];  // 字符串末尾终止符'\0'占一个字节
    strcpy(m_data, cstr);
  }
  else{
    m_data = new char[1];
    *m_data = '\0';
  }
}

inline
String::~String(){
  delete[] m_data;
}
```

这里我们写出的构造函数带有默认参数，其会在调用 String() 时，给出一个空字符串，而在以C风格字符串作为参数时，将该字符串复制一份给 m_data 。

在析构函数中，我们需要释放为 `m_data` 分配的空间。

拷贝构造函数 copy ctor

```
inline
String::String (const String & str){
  m_data = new char[strlen(str.m_data)+1];    // 这里可以直接调用另一个object的private变量
  strcpy(m_data, str.m_data);
}
```

拷贝构造函数用于处理类似 `String a(b)` 的函数调用，即使用 `b` 初始化 `a`。正如前面所提到的，如果忘记自定义拷贝构造函数，默认函数会将对象按照字节原封不动地复制一份，结果就是，`b` 和 `a` 的 `m_data` 会指向同一段内存空间，这显然不是我们想要的。所以，在上面的函数中处理了这个问题，为 `a` 分配了一段新的内存空间。

拷贝赋值函数 copy assignment operator

```
inline
String & String::operator=(const String& str){
  if(this == &str)   // 检测自我赋值
    return *this;

  delete[] m_data;
  m_data = new char[ strlen(str.m_data) + 1];
  strcpy(m_data, str.m_data);
  return *this;
}
```

拷贝赋值函数用于处理 `c = b` 的函数调用，即将 `b` 赋值给 `c`。那么考虑到class带有指针，在将b赋值给c时，需要先将 `c.m_data` 占用的空间释放，再为其分配一段新的空间。

此外，还要考虑“自我赋值”的特殊情况（一般情况下，不会有人故意写 `c = c` 这样的自我赋值，但是C++中还有着指针和引用这样的数据类型，可能会导致隐式的自我赋值）。

为什么返回 String &?

返回reference是为了支持连续赋值，比如 `a = b = c`，由于等号是从右到左结合的，因此相当于 `a = (b = c)`，这样 `b = c` 会返回 `b` 的reference，再赋值给 `a`。

但有些情况下，必须返回 `const String &`，比如我们为 `String` 定义了+运算符，代表将两个字符串拼接，调用方式为 `c = a + b`，那么加号运算符应当返回什么类型呢？如果返回 `String &`，那么诸如 `(a+b) = (c+d)` 的奇怪调用就是合法的了（为临时变量赋值没有意义），所以加号运算符的返回值必须是 `const String &`。

尽可能地使用 pass-by-reference 替换 pass-by-value

（参考自Effective C++ 条款20）

缺省情况下 C++ 以 by value方式传递对象至函数，也就是说，函数参数是以实际参数的副本为初值，而调用端获得的返回值也是函数返回值的一个副本。这些副本是由对象的 copy 构造函数产出的，这可能使得 pass-by-value 的操作造成额外的开销，正如我们上

节课所提到的那样，使用 by-reference 的方式，可以避免不必要的构造函数调用，从而减少开销，如果希望限定参数不被修改，则可以使用 pass-by-reference-to-const的方式。

这里我们介绍 by reference 的另一个优势，即避免 slicing （对象切割）问题。当一个 derived class （派生类）的对象以 by value 的方式传递，并被视为一个 base class 对象时，base class 的 copy 构造函数会被调用，而 derived class 的那些特化性质全被切割掉了，仅仅留下一个 base class 的对象（因为是base class 的构造函数生成的），这有些时候并不是我们想要的。

举例来说，假设你以前面介绍的 `String` 作为 base class，派生出一个 `BeautifulString`

```
class String{
public:
  ...
  virtual void display() const;  // 显示字符串
}

class BeautifulString{
public:
  ...
  virtual void display() const;  // 漂亮地显示字符串
}
```

现在你写了一个函数，用来显示字符串，下面是错误示范：

```
void displayString(String str){  // 不正确，参数被切割
  str.display();
}
```

当你调用上面的函数，并交给它一个 `BeautifulString` 对象，会发生什么事情呢？答案是使用 base class 的 `display()` 进行显示，而非使用 derived class 的 漂亮显示。对于 `displayString` 函数而言，不论传递过来的对象原本是什么类型，参数 `str` 就像一个 `String` 对象一样，多余的特性都被切割掉了，这与我们编程的想法有些出入。

解决的方法就是 by reference to const 的方式来传递 `str`：

```
void displayString(const String & str){  // 正确，参数不会被切割
  str.display();
}
```

现在，传进来的 `str` 是什么类型，就用哪种类型的 `display()` 进行展示。

栈stack与堆heap

Stack，是存在于某作用域的一块内存空间。例如当你调用函数，函数本身即会形成一个stack用来放置它所接收的参数，以及返回地址。在函数体内部声明的任何变量，所使用的的内存块都取自上述stack。

Heap，是指由操作系统提供的一块全局内存空间，程序可动态分配，从中获得若干区块。

```
Complex c3(1,2);

int main()
{
  Complex c1(1,2);
  static Complex c2(1,2);
  Complex *p = new Complex(3);
}
```

变量的生命周期

在上面的例子中：

c1是一个stack object，其生命周期在作用域结束之际结束。

c2是一个static local object，其生命周期在作用域结束之后仍然存在，直到整个程序结束。

c3是一个global object，其生命周期直到整个程序结束。

指针p本身是一个stack object，但是它所指向的是一个heap object，heap object的生命周期在它被delete之时结束。如果忘记`delete p`，则会造成内存泄漏。