第二课 类与对象

- 类的基本结构
- 类的基本成员
- 运算符的重载方法
- 友元函数和友元类
- 静态数据成员
- 引用成员
- 多文件编译

类

类是有着共同特征与行为、而状态各不相同的物体的总称。

- 对象是类的实现，是类的实例。
- 用C++术语来表达：类是一种类型。
- 将抽象出的数据成员、成员函数相结合，视它们为一个整体构成了类。

类 的 结 构

```
class cstring   // cstring是类名
{
public:             // 权限控制
// ...              // 公有成员
protected:          // 权限控制
// ...              // 保护成员
private:            // 权限控制
// ...              // 私有成员
};                  // 注意不要漏掉这个分号
```

类 的 结 构

```
class cstring   // cstring是类名
{
public:              // 请注意冒号
// 可以有多个public，也可一个也没有
// ...
protected:
// 可以有多个protected也可一个也没有
// ...
private:
// 可以有多个private，也可一个也没有
// ...
};
```

类 的 结 构

```
struct cstring   // struct和class用法几乎相同
{
public:
// 可以有多个public也可一个也没有
// …
protected:
// 可以有多个protected也可一个也没有
// …
private:
// 可以有多个private，也可一个也没有
// …
};
```

```
class cstring
{
public:
  typedef unsigned long size_type;
  static const size_type npos；
public:
  cstring(const char* other = "");
  cstring(const cstring&);
  ~cstring();
  cstring& operator = (const cstring&);
public:
  char& operator [ ] (size_type);
  bool operator == (const cstring&) const;
  bool operator < (const cstring&) const;
  cstring& operator += (const cstring&);
  friend cstring operator + (const cstring&, const cstring&);
  size_type size() const;
  // …
private:
  size_type m_sz;
  char* m_str;
};
```

```
class cstring
{
public:
  // 构造函数，用于构造一个对象
  cstring(const char* other = “”);          // 可以带默认值””
  // 拷贝构造函数，用于拷贝生成一个新对象
  cstring(const cstring&);                  // 注意它的参数书写形式
  // 析构函数，用于销毁过期的对象，释放资源
  ~cstring();
  // 赋值运算符=，用于以赋值的形式复制对象
  cstring& operator = (const cstring&); // 注意它的书写形式
  // …
private:
  // …
};
```

如果不给出任何定义，编译器将默认生成4个函数：构造函数，拷贝构造函数，析构函数，赋值运算符成员函数

```
class cstring
{
public:
  cstring() {}
  cstring(const cstring& s)  : m_sz(s.m_sz), m_str(s.m_str) {}
  ~cstring() {}
  cstring& operator = (const cstring& s)
  {
    m_sz = s.m_sz;
    m_str = s.m_str;
  }
private:
  size_type m_sz;
  char* m_str;
};
```

构造函数的定义

```
class cstring
{
public:
  typedef unsigned long size_type;
  static const size_type npos；
public:
  // 用成员初始化列表来初始化，初始化的顺序仅仅依赖于声明的顺序
  cstring(const char* other = "") : m_sz(std::strlen(other)),
      m_str(new char[m_sz+1])                    // 干嘛+1呀？
  {
    std::strcpy(m_str, other);
  }
  // …
private:
  size_type m_sz;
  char* m_str;
};
```

拷贝构造函数的定义

```
class cstring
{
public:
  // …
  cstring(const cstring&); // 拷贝构造函数的参数必须是常量引用类型！
  // …
private:
  size_type m_sz;
  char* m_str;
};

cstring::cstring(const cstring& other) : m_sz(other.m_sz),
                                         m_str(new char[m_sz+1])
{
  std::strcpy(m_str, other.m_str);
}
```

析构函数的定义

```
class cstring
{
public:
  // …
  ~cstring();
  // …
private:
  size_type sz;
  char* m_str;
};

// 做嵌入式，析构函数一般不要inline哦
cstring::~cstring() { delete [ ] m_str; } // 前面别加inline
```

赋值运算符的定义

```
cstring& cstring::operator = (const cstring& other)
{
  if(this != &other)  // 防止做a = a;这样的傻事
  {
    delete [ ] m_str;
    m_sz = other.m_sz;
    m_str = new char[m_sz+1];
    std::strcpy(m_str, other.m_str);
  }
  return *this;        // 注意这里有个*this
}

// 它也是一个成员函数，只不过习惯上称之为赋值运算符罢了
```

成员函数的定义

```
// 转换成C风格字符串
inline const char* cstring::c_str() const { return m_str; }

// 当前字符串里面含有多少个字符
inline cstring::size_type cstring::size() const { return m_sz; }

// 是空窜吗？
inline bool cstring::empty() const { return 0 == m_sz; }

// 清空字符串
inline void cstring::clear() {  m_str[m_sz = 0] = 0; }

// 注意在类外实现成员函数的书写格式！

// 实现都这么简单啊，所以干脆都让它们内联吧！
```

类 内 const 的 用 法

static const size_type npos = static_cast<size_type>(-1UL);

功能相当于：enum {npos = static_cast<size_type>(-1UL)};

const表示npos的值被定义成常数，禁止修改它。

在类内定义常量并直接初始化目前只有这两种方法合法。

cstring(const cstring& other);

由于拷贝构造函数的参数是个常量引用，它表示本函数承诺不会修改引用值。

const char& operator [ ] (size_type i) const;

这是一个对运算符[ ]的重载函数，第一个const表示返回值不允许修改，第二个const表示本成员函数承诺不会修改当前类的数据成员。与之相对应的非常量版本是下面这个：

char& operator [ ] (size_type i);

通过这个非常量版本可以修改一个对象所存储字符串的某个字符。

一般而言，上面的那个常量版本用不上，定义它的目的是防止有些编译器拒绝编译。

运 算 符 的 重 载

C语言里面有运算符重载吗，谁能告诉我？

运算符的重载语法格式：

返回类型 operator 运算符(参数列表)

当没有返回值时，返回类型用void，关键字operator表示后面出现的将是运算符，它是必需的。参数列表根据实际是否需要可能0个或多个，小括弧必不可少不论有多少个参数。

注意有几个运算符不能重载。

::、 .、.* 、?:、 sizeof、 typeid、强制类型转换运算符(dynamic_cast, const_cast, static_cast, reinterpret_cast)。

为何C++不允许重载他们？

运算符重载的一般性原则是：别把自己弄糊涂。例如：把加号重载成减号语法上虽然没问题，但可能给理解造成困惑。

运算符重载的意义：比书写函数形式上更简洁。

几个运算符的重载

```
inline char& cstring::operator [ ] (size_type i) { return m_str[i]; }
// 运算符[ ]只能使用成员函数来重载，只能通过类成员函数重载的运算符有[ ]、()、->等等
inline const char& cstring::operator [ ] (size_type i) const { return m_str[i]; }

// 允许用==判断两个字符串是否相等
bool cstring::operator == (const cstring& other) const
{
    if(m_sz != other.m_sz)                    // 能省点开销则省点
        return false;
    else
        return 0 == std::strcmp(m_str, other.m_str);
}
// 允许两个字符串用<比较大小
bool cstring::operator < (const cstring& other) const
{
  return std::strcmp(m_str, other.m_str) < 0;  // strcmp?
}
```

几个运算符的重载

```
// 更多比较运算符的重载
bool operator != (const cstring& other) const { return !(*this == other); }
bool operator > ( const cstring& other) const { return other < *this; }
bool operator <= (const cstring& other) const { return !(other < *this); }
bool operator >= (const cstring& other) const { return !(*this < other); }
```

由此看来，看来我们只需要重载==和<就足够了。想一想数学上怎么定义==和<的?

进一步研究发现其实只需要定义<就够了，因为在数学上如果:

a <= b且a >= b则可以推理得出a == b

只不过在实际应用中我们需要考虑很多种因素(例如：时间和内存开销)，我们常常一次性重载==和<，其它几个比较运算符的重载实现则可通过这两个运算符简单拼凑出来。

这里涉及到一点泛型基础，关于这个问题在将来讲模板和泛型时会更深入讨论。

几个运算符的重载

```
// 允许直接用+把第2个字符加到第1个字符串上
cstring& cstring::operator += (const cstring& other)        // 请注意返回值类型
{
  char* pstr = new char[m_sz+other.m_sz+1];
  std::strcpy(pstr, m_str);
  std::strcpy(pstr+m_sz, other.m_str);
  m_sz += other.m_sz;
  delete [] m_str;
  m_str = pstr;
  return *this;
}

// 允许在一个字符串后追加一个字符
cstring& cstring::operator += (char ch)
{
  char* pstr = new char[m_sz+2];
  std::strcpy(pstr, m_str);
  pstr[m_sz++] = ch;
  delete [] m_str;
  m_str = pstr;
  return *this;
}
```

几个运算符的重载

```
// 这个声明看上去有点与众不同哦
friend cstring operator + (const cstring&, const cstring&);

// 在类外的实现里，operator前面怎么没cstring::啊?
cstring operator + (const cstring& str1, const cstring& str2)
{
  cstring tmp_str(str1);
  tmp_str += str2;          // 这里调用了谁?
  return tmp_str;
}
```

什么时候设计成+=，什么时候设计成+？注意返回值类型。

friend又是干什么用的？

友　　元

友元有三种，友元函数，友元类和友元成员函数 (友元成员函数不讲) 。

先看看友元函数

友元函数：友元函数是类成员，故在类外实现时前面不加friend关键字。友元函数可以直接访问私有成员，而不论它在类里面什么地方声明。

```
// 一个友元函数的声明，需要重载<<
friend std::ostream& operator << (std::ostream&, const cstring&);
// 对运算符<<重载的实现，用于直接输出一个字符串的值
std::ostream& operator << (std::ostream& os, const cstring& str)
{
  if(!str.empty())
    os << str.m_str;
  return os;                  // 别忘了返回这个引用，否则不能连续输出哦
}
```

如果想从键盘终端设备上读入一个字符串该怎么重载运算符>>呢？(练习)

友元类举例

```
#include <iostream>
class A
{
  friend class B;                    // B是A的友元
public:
    void Display()  { std:: cout << x << '\n';}
private:
    int x;
};
class B
{
public:
    void Set(int i);
    void Display();
private:
    A a;
};
```

```
void B::Set(int i)
{
    a.x = i;        //访问组合对象的私有成员正是友元的“特长”
}

void B::Display()
{
    a.Display();
}

int main()
{
    B b;
    b.Set(1000);
    b.Display();
}
```

友元类关系的性质

友元关系是单向的

如果B类是A类的友元，则B类的函数可以访问A类的私有和保护数据，但A类的成员函数却不能访问B类的私有、保护数据

友元关系不能传递

如果B类是A类的友元， C类是B类的友元，若没特别声明，则 C类和A类无友元关系。

友元关系不能继承

如果B类是A类的友元， B类的子类不会自动成为A类的友元类。

静态整型常量数据成员

有一个静态数据成员npos，它怎么初始化呢？

对于普通的常量，声明时直接初始化。类中的常量成员不能在声明时直接初始化。但是，对于静态整型常量，可以直接写在类内初始化：

```
static const size_type npos = static_cast<cstring::size_type>(-1UL);
```

请记住以上用法的条件是：静态整型常量，这三个限制条件缺一不可。

当然也可以在类外这样初始化：

```
const cstring::size_type npos = static_cast<cstring::size_type>(-1UL);
```

实现同样的功能，还有第二种解决办法，用枚举来表示：

```
enum {npos = static_cast<size_type>(-1UL)};
```

如果仅仅是静态数据成员，并非什么整型常量，那只好在类内声明，在类外进行初始化了，没得商量。

值得注意的是：不论用一个类创建多少个对象，静态成员只能公用同一个。

对象的定义

- 类的对象是该类的某一特定实体，即类类型的变量。
- 定义对象时，系统会为每一个对象分配自己的存储空间，该空间只保存数据，函数代码是所有对象共享的。
- 声明形式：

  类名　对象名;
- 例：

  cstring str1, str2;

成员的访问方式

- 所谓对成员的访问，是为成员的行为立的规矩，是封装性的体现。
- 有两种方式：

1. 类内访问是类中成员的相互访问 ，又称成员访问

   用法：直接使用成员名。

2. 类外访问是类的外界访问类中的成员，又称对象访问。是在类外访问类的public 属性的成员。

用法：使用“对象名.成员名”

例如：

```
cstring str1(“1234567890”), str2 = str1, str3;
str1.swap(str3);
std:cout << (str1 += str2) << ‘\n’;
std::cout << str2.size() << ‘\n’;
// …
```

构造函数的调用

调用构造函数将导致开辟一定的内存并创建一个对象，例如：

cstring my_str(“0123456789”);

该语句将执行代码：

```
cstring(const char* other = "") : m_sz(std::strlen(other)),
                                  m_str(new char[m_sz+1])
{
  std::strcpy(m_str, other);
}
```

创建一个对象**my_str**，创建过程中计算出字符个数**10**存储在栈上,字符串内容**“0123456789”**则存储在堆上。

拷贝构造函数的调用

调用拷贝构造函数将导致开辟一定的内存并创建一个对象，
例如：

```
cstring new_str(my_str); // cstring new_str = my_str;
```

该语句将执行代码：

```
cstring(const cstring& other) : m_sz(other.m_sz),
    m_str(new char[m_sz+1])
{
  std::strcpy(m_str, other.m_str);
}
```

创建一个对象new_str，它与my_str的内容完全相同。

析构函数的调用

对象过期时，析构函数会自动调用，无需程序员人为干预。例如：

```
int main()
{
  cstring str1;
  {
    cstring str2(“0123456789”);
  } // 遇到这个}时，str2过期，调用析构函数一次

  return 0;  // 遇到return时，str1过期，调用析构函数一次
}
```

注意：最好不要显式调用构造函数(像str1.~cstring())。

赋值运算符的调用

什么时候调用赋值运算符=呢？

一个对象给另外一个对象赋值时，例如：str2 = str1;

这条语句将执行这些代码：

```
cstring& cstring::operator = (const cstring& other)
{
    if(this != &other)
    {
        delete [] m_str;
        m_sz = other.m_sz;
        m_str = new char[m_sz+1];
        std::strcpy(m_str, other.m_str);
    }
    return *this;
}
```

静态常量的调用方法

cstring有个公有的静态整型常量，可以类似如下这样调用：

std::cout << cstring::pos; // 显示它的大小

因为静态数据成员不在对象中创建，而为一个类定义的所有对象公用。

如果常量是枚举类型，也可以类似以上那样调用。

了解性内容：

静态常量存储在什么地方？

栈上，堆上，还是…？

引用成员的初始化

```
class A
{
public:
    A(int i = 0);
    operator int () const;
protected:
    const int& r;
private:
    const int a;
};

A::A(int i) : a(i), r(a) { }
```

// 引用作类的数据成员只能通过初始化列表来获得初值 .
// 这也就要求我们：必须显式地给出构造函数。

```
A::operator int() const { return a; }

#include <iostream>

int main()
{
    A a(100);
    std::cout << "value of a: " << a << '\n';
    std::cout << "size of a: " << sizeof(A) << '\n';
};

// 编译一下，看看提示什么信息，为什么不出错？(练习)
```

类型转换

语法格式：

operator 类型 () const

其中**operator**必不可少，类型就是你希望要转换成的类型。

看下面这个例子：

```
A::operator int() const { return a; }
```

因为定义了类型转换成员函数，所以下面的语句得以正确执行。

```
int main()
{
   A a(100);
   std::cout << “value of a: ” << a << ‘\n’; // 这个a可是A类型的哦！
   // …
};
```

使用条件编译的头文件

多文件结构

```
//head.hpp
#ifndef  HEAD_HPP   //避免递归包含的宏
#define  HEAD_HPP
//    …
class A
{
// …
};
// …
#endif
```

使用条件编译的头文件

多文件结构

```
//head.hpp
#pragma once  // 这样也可以达到同样目的
//    ...
class A
{
// ...
};
// ...
```

多文件环境中变量的使用

- 多个文件中可以多次声明变量，但只能定义一次；
- 所谓全局变量是指跨文件的全局之意，在一个文件中定义了一个全局变量，在别的文件中不可再定义同名的变量，只能用**extern**使用它；或用**static** 各自定义自己的“全局变量”；
- 上面讲的变量对类同样适用。（只写类头叫类声明，带类体叫类定义）；
- 仅声明的类不能定义对象，只能定义类的指针或引用，定义的类才能定义对象。

多文件环境中函数的使用

- 函数声明是只写函数原形的语句；
- 函数定义(实现)是带函数体的函数原形语句；
- 多个文件中可以多次声明函数，但只能定义一次；
- 不用**extern**，函数自然是可以跨文件使用的（默认）；
- 加上**static**则意味着该函数仅供本文件使用；

编程练习

1. 本次课例子cstring中有些成员函数没有给出实现，请你补全所有实现，并编写测试代码测试整个cstring的正确性，如果发现错误请改正。

2. 设计一个单链表slist用来处理浮点型数据。下面给出该类的所有成员声明(如果这些声明或成员名字不符合你的习惯，请自行定义)。

```
// 单链表节点
struct snode
{
  double data;
  snode* next;
  snode(const double& d = 0.0, snode* n = 0) : data(d), next(n) {}
};  // snode
```

```
// 一个单链表的类
class slist
{
private:
  snode* head;   // 表头指针，空表表头指针值为0

  // 返回ptr指向节点的前一个节点指针, 如果ptr == head则返回0
  // 否则返回ptr指向节点的前一个节点指针
  snode* previous(snode* ptr)
public:
  typedef double value_type;

  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;

  typedef const value_type& const_reference;
  typedef std::size_t size_type;              // 注意需要 #include <cstddef>
 public:
```

```
slist ();                          // 构造函数
// 显式构造函数,使用元素x构造含有n个节点的一个链表
explicit slist (size_type n, const_reference x = value_type ());
slist (const slist& sl);   // 拷贝构造函数
~slist();                          // 析构函数
slist& operator =(const slist& sl);      // 重载赋值运算符

void push_front(const_reference x); // 在链表头前插入一个元素x
void push_back(const_reference x); // 在链表尾后插入一个元素x
void pop_front();                                  // 删除链表的第1个节点
void pop_back();                                  // 删除链表最后一个节点

void clear ()；                // 清空整个链表
// 在p指向节点的前面插入一个元素x
void insert(snode* p, const_reference x);
// 在p指向节点的后面插入一个元素x
insert_after(snode* p, const_reference x);
// 删除p指向的当前节点
void erase(snode* p);
// 删除p指向节点相邻后面的那个节点
void erase_after(snode* p);
```

```
  void reverse();        // 逆转链表中的每个节点

  // 判断链表是否为空，如果为空则返回值为真，否则假
  bool empty () const;

  size_type size() const; // 返回链表中存储的元素个数

  const_reference front() const; // 返回链表第一个节点的值

  reference front();                  // 返回链表第一个节点的值

  void swap(slist& sl);               // 把sl和和当前链表交换

  snode* header() const;              // 返回表头节点指针
};

// 逐个比较两个单链表中存储的数据，如果相等则返回值为真，否则为假
bool operator == (const slist& sl1, const slist& sl2);

//逐个比较两个单链表中存储的数据，如果前者较小返回值为真，否则为假
bool operator < (const slist& sl1, const slist& sl2);
```

3. 自己设计一个栈，处理数据类型为整型数据，包括声明，实现和测试。
4. 设计一个普通队列，处理数据类型为整型数据，包括声明实现和测试。

Thanks!