C++编程(8)

Tang Xiaosheng
北京邮电大学电信工程学院

第十章 类

- 引言
- 类
- 高效的用户定义类型
- 对象
- 忠告

10.1 引言

- C++里类的概念的目标是为程序员提供一种建立新类型的工具，使这些新类型的使用能够象内部类型一样方便
- 类型是某个概念的一个体现，类其实就是一种用户定义类型
- 如果一个程序里提供的类型与应用中的概念有直接的对应，与不具有这种情况的程序相比，这个程序很可能就更容易理解，也更容易修改

引言

- 定义新类型的基本思想就是将实现中那些并非必然的细节(例如，用户存储该类型的对象所采用的数据的布局)，与那些对这个类型的正确使用至关重要的性质(例如，能够访问其中数据的完整的函数列表)区分开来
- 表达这种划分的最好方式就是提供一个特定的界面，对于数据结构以及内部维护例程的所有使用都通过这个界面进行

10.2 类

- 成员函数
- 访问控制
- 构造函数
- 静态成员
- 类对象的复制
- 常量成员函数
- 自引用
- 结构和类
- 在类内部的函数定义

10.2.1 成员函数

- 考虑用struct实现日期
- 改进版本，将函数加入到struct中
- 在一个类中声明的函数叫做成员函数(struct也是一种类)

```
struct Date {
    int d,m,y;
}
void init_date(Date&  d, int, int, int);
void add_year(Date&  d,int n);
void add_month(Date&  d, int n);
void add_day(Date&  d, int n);
// 函数和日期类型之间没有任何显式的联系
```

```
struct Date {
    int d,m,y;
    void init_date(Date&  d, int, int, int);
    void add_year(Date&  d,int n);
    void add_month(Date&  d, int n);
    void add_day(Date&  d, int n); }
// 改进版本
```

成员函数

- 成员函数只能通过适当类型的特定变量，采用标准的结构成员访问语法形式调用

```
Date my_birthday;
void f() {
    Date today;
    today.init(16,10,1996);
    my_birthday.init(30,12,1950);
    Date tomorrow = today;
    tomorrow.add_day();
    //…
};
```

```
//定义成员函数时使用的方式
void Date::init(int dd,
        int mm, int yy)
{
    d = dd; // d是结构内部的成员
    m = mm;
    y = yy;
}
// d m y 是函数被调用时所针对的
那个对象的成员
```

类定义

- class X { ... };
  被称为一个类定义，因为它定义了一个新类型
- 由于历史的原因，一个类定义也常常被说成是一个类声明
- 就像其他不是定义的声明一样，类定义可以由于#include的使用而在不同的源文件里重复出现，这样并不违反唯一定义规则

10.2.2 访问控制

- 通过struct Date声明，提供了一组操作Date的函数，然而，它却没有清楚地说明这些函数就是直接依赖于Date表示的全部函数，而且也是仅有的能够直接访问Date类的对象的函数
- 这些限制可以通过用class替代struct而描述清楚

访问控制

标号public将这个类内部分成两个部分，其中第一部分(私用部分)只能由成员函数使用；而第二部分(共用部分)则构成了该类的对象的公用界面，struct也是class，不过其成员的默认方式是共用的

```
class Date {
    int d, m, y;
public:
    void init (Date& d, int, int, int);

    void add_year(Date& d,int n);
    void add_month(Date& d, int n);
    void add_day(Date& d, int n);
};
```

```
// 函数定义同前
void Date::add_year(int n)
{
    y += n;
}
```

访问控制

```
//非成员函数将禁止访问私有成员
void timewrap(Date& d)
{
    d.y -= 200; // Error
}
```

这种限定的好处是：

1 导致日期异常的错误必然是由某个成员函数造成的

2 一个潜在的用户要学习使用一个类也只需要考察其成员函数的定义

该类中init函数的原因

1 有一个函数来设置对象的值总是有用的

2 数据的私用性也要求我们这么做

10.2.3 构造函数

- 采用init一类的函数提供对类对象的初始化，这么做既不优美又容易出错(比如说，忘记调用)
- 更好的途径是让程序员有能力去声明一个函数，其明确目的就是去完成对象的初始化，此类函数称为构造函数
- 构造函数具有与类相同的名字

构造函数

```
class Date{
    //…
    Date(int, int, int);
    //构造函数 Constructor
    // ctor
};
```

若一个类有构造函数，这个类的所有对象的初始化都将通过对某个构造函数的调用完成初始化，注意参数匹配问题，可以创建多个构造函数来以多种方式完成对象的初始化工作

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);
// 简写形式
Date my_birthday;
// Error 缺少初始式
Date release1_0(10,12);
// Error 三个参数
```

```
class Date {
    int d, m, y;
public:
    Date(int, int, int);
    Date(int, int); // 构造函数的重载
    Date(int);
    Date();
}
```

构造函数

```
class Date {
    int d, m, y;
public:
    Date(int dd=0,
         int mm=0, int yy=0);
    //…
};
Date::Date(int dd, int mm, int yy)
{
    d = dd? dd : today.d;
    m = mm? mm : today.m;
    y = yy? yy : today.y;
}
```

也可以用带默认参数的函数来取代函数重载

10.2.4 静态成员

- 为Date提供默认值的方式隐藏着一个重要问题：Date类现在依靠一个全局变量today
- 如果设定一个这样的变量，而只在Date类中使用，用户会遇到很多不愉快的事情，而且维护工作也变得麻烦
- 对此的解决方法是：创建一个static静态成员，可以获得全局变量的方便，而又无须受到访问全局变量之累
- 对于所有对象，静态成员只有唯一的一个副本
- 使用一个静态成员，不必提及任何对象

静态成员

```
class Date{
    int d, m, y;
    static Date default_date;
public:
    Date(int dd=0,
            int mm=0, int yy=0);
    //…
    static void set_default(int, int, int);
};

Date::Date(int dd, int mm, int yy)
{
    d = dd? dd : default_date.d;
    m = mm? mm : default_date.m;
    y = yy? yy : default_date.y;
}
```

```
void f()
{
    Date::set_default(4,5,1945);
}

// 静态成员和函数需要在某个地方另行定义
Date
Date::default_date(16,12,1770);
void Date::set_default(int d,
        int m, int y)
{
    Date::default_date =
                Date(d,m,y);
}
//Date()相当于Date::default_date
```

10.2.5 类对象的复制

- 按照默认约定，类对象可以复制，例如：Date d = today; // 复制初始化
- 按照默认方式，类对象的复制就是其中各个成员的复制，如果某个类所需要的不是这种默认方式，那么就可以定义一个复制构造函数X::X(const X&)，由它提供所需要的行为
- 类似的，类成员也可以通过赋值进行按照默认方式的复制，例如：Date d; d = today;
- 对于赋值复制，默认语义是按照成员复制，用户可以定义合适的赋值运算符来完成自己需要的复制

10.2.6 常量成员函数

- 可以为Date类提供一些函数用来检查一个Date的值
- 注意，函数参数表后面的const，表示该函数不会修改Date的状态

```
class Date{
    int d, m, y;
public:
    int day() const {return d;}
    int month() const;
    int year() const;
}
inline int Date::year() const
{
    return y;
    // return y++; Error
}
```

```
void f(Date& d, const Date& cd)
{
    int i = d.year(); //ok
    d.add_year(1); // ok
    int j = cd.year(); // ok
    cd.add_year(1); //Error
}
// const或者非const对象都可以调用const成员函数，而非const成员函数则只能由非const对象调用
```

10.2.7 自引用

- 状态更新函数：add_year(),add_month(), add_day()被定义为不返回值的函数对于这样一组相关的更新函数，可以让他们返回一个到被更新的引用，以使对于对象的操作可以串接起来

```
class Date{
   // …
   Date& add_year(int n);
   Date& add_month(int n);
   Date& add_day(int n);
};
//d.add_year(1).add_month(1).
add_day(1)
// 可以如上述方式使用
```

```
Date& Date::add_year(int n)
{
   if(d==29 && m==2 &&
            !leapyear(y+n)){
      d = 1;
      m = 3;
   }
   y+=n;
   return *this;
}
```

关于this

- 在一个非静态的成员函数里，关键字this是一个指针，指向该函数的当时这次调用所针对的那个对象
- 在类X的非const成员函数里，this的类型就是X*，但是，不能取得this的地址或者给它赋值
- 在类X的const成员函数里，this的类型是const X*，以防止对这个对象本身的修改
- 大部分对于this的应用都是隐含的，特别的，对于一个类中的非静态成员的引用都要依靠隐式地使用this，以获取相应对象的成员

this示例

```
Date& Date::add_year(int  n)
{
    if(this->d==29 && this->m==2
         && !leapyear(this->y+n)){
        this->d = 1;
        this->m = 3;
    }
    this->y+ = n;
    return *this;
}
// 使用this的例子
```

```
Date::Date(int  y, int m, int d)
{
    this->y = y;
    this->m = m;
    this->d = d;
}
// 另外一种不得不使用this的例子
```

10.2.7.1 物理的和逻辑的常量性

- 偶尔有这种情形，一个成员函数在逻辑上是const，但它却仍然需要改变某个成员的值，对于用户而言，这个函数看似没有改变其对象的状态，然而，它却可能更新了某些用户不能直接访问的细节，这通常被称为逻辑的常量性
- 例如：Date类可能有一个函数，返回日期的字符串表示，构造字符串是相对费时的，实现中可以为其保留一个副本，重复需要时返回该副本即可

逻辑的常量性

```
class Date{
    bool cache_valid;
    string cache;
    void compute_cache_value();
    // …
public:
    string string_rep() const;
    //但是第一次调用string_rep时
    //需要构造cache字符串
}
```

```
string Date::string_rep() const
{
    if(cache_valid == false){
        Date* th =
            const_cast<Date*>(this);
        th->compute_cache_value();
        th->cache_valid = true;
    }
    return cache;
}
```

以上解决方式一点不优美，也无法保证总能工作，例如

```
Date d1;
const Date d2;
string s1 = d1.string_rep(); // ok
string s2 = d2.string_rep(); //无定义行为
```

10.2.7.2 可变的—mutable

- 为了解决上述显式“强制去掉const”的问题，可以将所涉及的数据声明为mutable

```
class Date{
    mutable bool cache_valid;
    mutable string cache;
    void compute_cache_value();
    // …
public:
    string string_rep() const;
}
```

```
string Date::string_rep() const
{
    if(cache_valid == false){
        compute_cache_value();
        cache_valid = true;
    }
    return cache;
}
```

```
Date d1;
const Date d2;
string s1 = d1.string_rep(); // ok
string s2 = d2.string_rep(); // ok
```

以上技术使得前面string_rep()的所有使用都合理化了

延迟求值(lazy evaluation)

- 若在某个表示中只有一部分允许改变，将这些成员声明为mutable式最合适的，但是如果有大批量都需要修改，最好将这些数据放入另外一个独立的对象里，并间接的访问它

```
struct cache{
    bool valid;
    string rep;
};
class Date{
    cache* c;
    void cache_value() const;
public:
    string string_rep() const;
};
```

```
string Date::string_rep() const
{
    if(!c->valid) {
        cache_value();
        c->valid = true;
    }
    return c->rep;
}
```

这种技术可以推广到各种方式的延迟求值

10.2.8 结构和类

- 按照定义，一个struct就是一个类，但其成员默认为公用的，也即：struct s{ … 是class s { public: … 的简写形式
- 一般情况下，struct被用于所有成员都是公用的那些类
- 这样的类并不是完整的类型，不过是个数据结构
- 结构中也可以有构造函数

```
class Date1 {
    int d, m, y;
public:
    Date1(int dd, int mm, int yy);
    void add_year(int n);
};
```

```
struct Date2{
private:
    int d, m, y;
public:
    Date2(int dd, int mm, int yy);
    void add_year(int n);
};
```

上述两个声明除了名字不同，完全等价

10.2.9 类内部定义的函数定义

- 如果一个函数是在类定义内部定义的(不是简单的声明)，那么这个函数就被做为内联函数处理
- 也就是说，在类内部定义的函数应该是小的，频繁使用的函数

```
class Date{
    int d, m, y;
public:
    int day() const
    {
        return d;
    }
}; // 函数和数据谁在前无所谓
```

<==>

```
class Date{
public:
    int day() const;
private:
    int d, m, y;
};

inline int Date::day() const
{  return d;  }
```

10.3 高效的用户定义类型

- 前面所提及的是设计一个Date类的各种细节，本部分主要内容讨论设计一个简单而高效的Date类，并阐明语言的各种特性来如何支持这种设计
- 应用系统中会频繁的使用很多种抽象，例如：字母、整数、复数、向量、数组等，C++直接支持其中一些最常见的抽象，但是，C++提供了使用户定义这些类型的机制

一个更好的Date类

```
class Date{
public:
    enum Month{ jan=1, feb, mar, apr,
may, jun, jul, aug, sep, oct, nov, dec};
// Month主要对付记忆的问题
// 欧美表示顺序不同，顺序反时能被检测出来
    class Bad_date{}; // 异常时抛出
    Date(int dd=0, Month mm=Month(0),
int yy=0); // 构造函数，注意日、月、年顺序
// 获取Date中数值的函数
    int day() const;
    Month month() const;
    int year() const;
    string string_rep() const;
    void char_rep(char s[]) const; // C风格
    static void set_default(int, Month, int);
// 修改Date的函数
    Date& add_year(int n);
    Date& add_month(int n);
    Date& add_day(int n);

private:
    int d, m, y;
    static Date default_date;
};

Date Date::default_date(22,
jan, 1901);
```

使用Date类示例

```
void f(Date& d)
{
    Date lvb_day = Date(16, Date::dec, d.year());
    if(d.day() == 29 && d.month() == Date::feb)
    {
        // …
    }
    if(midnight()) d.add_day(1);
    cout << “day after:” <<d+1 << ‘\n’;
}
```

实现Date类而不是简单的使用Date结构体可以使得Date的概念局部化，否则每个用户不得不直接操作Date的成分，会使得Date的概念散布到整个系统中，最终使得程序难于理解

10.3.1 成员函数

```
Date::Date(int dd, Month mm, int yy)
{
    if(yy==0) yy=default_date.year();
    if(mm==0)
        mm=default_date.month();
    if(dd==0) dd=default_date.day();

    int max;
    switch(mm) {
    case feb:
        max = 28 + leapyear(yy);
        break;
    case apr: case jun: case sep: case
nov:
        max = 30;
        break;
     case jan: case mar: case may:
case jul: case aug: case oct:
case dec:
        max = 31;
        break;
    default:
        throw Bad_date();
    }
    if(dd<1 || max<dd)
        throw Bad_date();
    y = yy; m = mm; d = dd;
}
// 构造函数完成构造一个合法的日期，
一旦完成，其他函数就只需要用来对
其赋值和更改，而不需要合法性检查
了，这样能极大地简化代码
```

10.3.2 协助函数

- 一个类可能有一批与它相关的函数，但是又没有必要定义在类里，传统方式下，它们的声明将直接与类Date的声明放在同一个文件里(一般为.h文件)，此外，我们也可以将这个类和它的协助函数包在同一个名字空间里
- 一般情况下，一个名字空间里的内容比下述Chrono要多的多

```
namespace Chrono{
    class Date { /*…*/ };
    int diff(Date a, Date b);
    bool leapyear(int y);
    Date next_weekday(Date d);
    Date next_saturday(Date d);
}
```

10.3.3 重载的运算符

- 增加一些具有习惯形式的函数很有用，比如，使用==来判断两个Date对象是否相等

```
bool operator==(Date a, Date b)
{
    return a.day() == b.day() && a.month() == b.month()
           && a.year() = b.year();
}
```

类似的：

```
bool operator!=(Date, Date);
bool operator<(Date, Date);
bool operator>(Date, Date);
Date& operator++(Date& d);
Date operator+(Date d, int n);
```

10.3.4 具体类型的意义

- 像Date这样的简单用户定义类型称作具体类型，是为了让它们与抽象类型与类层次结构区别，同时也是为了强调这种类型与内部int或者char等内部类型的相似性
- 这种类型有时也被称为值类型
- 定义良好的一组这种类型(小而有效)能够为应用提供一个基础，提高程序员开发效率

10.4 对象

- 对象的建立有多种可能，有的对象是局部变量，有的是全局变量，有的是类的成员等，本部分讨论这方面的不同情况、统辖着它们的规则、初始化对象所用的构造函数，以及在它们变得不可用之前清理他们的构造函数

10.4.1 析构函数

- 构造函数完成对象的初始化，因此，在类中需要一个函数完成对象的清理工作，并且该函数能够象构造函数一样被自动调用，这个函数称为析构函数(deconstructor)
- 析构函数通常完成一些清理和释放资源的工作，一般情况下，用户不需要显式地调用它
- 对于没有析构函数的类型，可以简单认为这种类型有一个什么都不做的析构函数

析构函数示例

```
class Name{ const char* s; };
class Table{
    Name* p;
    size_t sz;
public:
    Table(size_t s = 15) { p = new Name[sz=s]; }
    ~Table() { delete []p; }
    Name* lookup(const char*);
    bool insert(Name*);
};
```

10.4.2 默认构造函数

- 可以认为大部分类型都有一个默认构造函数
- 默认构造函数就是调用时不必提供参数的构造函数，上例中：Table::Table(size_t)就是默认构造函数
- 若用户没有声明构造函数，并且有必要，编译器会设法去生成一个
- 编译器生成的默认构造函数将隐式地为类类型(class type)的成员和基类调用有关的默认构造函数

默认构造函数

```
struct Tables{
    int i;
    int vi[10];
    Table t1;
    Table vt[10];
};
Tables tt;
```

tt将会被用一个生成出来的默认构造函数初始化，该构造函数为tt.t1以及tt.vt的每个成员调用Tables(15)，但是它不会去初始化tt.i和tt.vi，因为它们不是class type

```
struct X{
    const int a;
    const int& r;
};
X x; // Error
```

由于const和引用必须进行初始化，包含const或者引用成员的类就不能进行默认构造

此外，默认构造函数也可以被显示调用(10.4.10)，内部类型也有默认构造函数

10.4.3 构造和析构

- 对象可以有各种不同方式
- 10.4.4 一个命名的自动对象
- 10.4.5 一个自由存储对象
- 10.4.6 一个非静态成员对象
- 10.4.7 一个数组元素
- 10.4.8 一个局部静态对象
- 10.4.9 一个全局对象
- 10.4.10 一个临时对象
- 10.4.11 一个在用户函数获得的存储里放置的对象
- 10.4.12 一个union成员

10.4.4 局部变量

- 对一个局部变量的构造函数将在控制线程每次通过该变量的声明时执行
- 每次当控制离开该局部变量所在的块时，就会去执行它的析构函数
- 局部变量的析构函数将按照构造它们的相反顺序执行

```
void f(int i)
{
    Table aa;
    Table bb;
    if(i>0) { Table cc;  }
    Table dd;
}
```

10.4.4.1 对象的复制

- 对象复制的默认含义就是对象按照成员逐个复制到另外一个中去
- 右边例子中，构造函数被调用了2次，而析构函数被调用了3次
- 为避免这类情形，可以将Table复制的意义定义清楚(复制构造函数，复制赋值)

```
void h()
{
    Table t1;
    Table t2 = t1;
    // 复制初始化
    Table t3;

    t3 = t2;
    // 复制赋值
}
```

对象的复制

```
class Table
{
    // ...
    Table(const Table&);
    Table& operator= (const
        Table&);
};

Table::Table(const Table& t)
{
    p = new Name[sz=t.sz];
    for(int i=0;i<sz;i++)
        p[i]=t.p[i];
}
```

```
Table& Table::operator=(const
Table& t)
{
    if(this != &t) { // 防止自赋值
        delete []p;
        p = new Name[sz=t.sz];
        for(int i=0;i<sz;i++)
            p[i]=t.p[i];
    }
    return *this;
}

// Table t1;
// Table t2 = t1;    // 复制初始化
// Table t3;
// t3 = t2;  // 复制赋值
```

10.4.5 自由存储

- 使用new的方式建立对象时，这些对象在自由存储里面，该对象将一直存在直到将delete函数用于指向它的指针
- 有些编程语言可以自动进行内存管理

```
int main() {
    Table* p = new Table;
    Table* q = new Table;
    delete p;
    delete p; // 行为没有定义
}
```

10.4.6 类对象做为成员

```
class Club{
    string name;
    Table members;
    Table officers;
    Date founded;
    Club(const string &,
        Date fd);
};
Club::Club(const string & n,
    Date fd): name(n),
    member(),officers(),
    founded(fd)
{    // 上面的member()
}    // 和officers()也可无
```

- 包含类成员的类的构造函数
- 成员初始式列表由一个冒号开头，用逗号分割
- 成员的构造函数将在容器类的构造函数前被执行**(严格按照在类中声明的顺序执行)**
- 成员类的析构函数后于容器类的析构函数而执行，并按照与构造时相反的顺序逐个被调用

10.4.7 数组

- 如果在构造某个类的对象时可以不必提供显式的初始式，那么就可以定义这个类的数组
- 例如：Table tbl[10];
- 将建立一个10个Table的数组，并用默认参数15调用Table::Table()，对每个Table进行初始化
- 也可以使用指针，例如：Table* p1 = new Table[10];
  注意此时需要用delete[] p1来回收空间
- 而Table *p1=new Table;只需要delete p1;

10.4.8 局部静态存储

- 对于局部静态对象，构造函数是在控制线程第一次通过该对象的定义时调用
- 局部静态对象的析构函数将按照它们被构造的相反顺序逐一调用，没有规定确切的时间

```
void f(int i)
{
    static Table tbl;
    if(i) {
        static Table tbl2;
    }
}
```

```
int main()
{
    f(0);
    f(1);
    f(2);
}
```

10.4.9 非局部存储

- 在所有函数之外定义的变量(即全局变量、名字空间的变量，以及各个类的static变量)，在main()被激活之前完成初始化(构造)
- 对于已经构造起的所有这些变量，其析构函数将在退出main()之后调用
- 在一个编译单位里，对非局部变量的构造将按照它们的定义顺序进行，析构函数则按相反顺序逐个被调用
- 不同编译单位的非局部变量，构造和析构的顺序没有保证

非局部存储示例

```
class X{
    // …
    static Table memtbl; // 注意这是个声明
};
Table tbl; // 第一个被构造
Table X::memtbl; // 第二个被构造
namespace Z{
    Table tbl2; // 第三个被构造
}
```

10.4.10 临时对象

- 临时对象最经常是作为算术表达式的结果出现的，例如：x*y+z过程中的某一点
- 除非一个临时对象被约束到某个引用，或者被用于做命名对象的初始化，否则它将总在建立它的那个完整表示式结束时销毁
- 完整表达式就是那种不是其他表达式的子表达式的表达式

临时对象示例

```
void f(string& s1, string& s2, string&
s3)
{
    const char* cs = (s1+s2).c_str();
    // s1+s2被保存为临时对象
    // 表达式运算完毕后会被删除
    // 这样，下面的cs输出很难保证正确
    // 因为指向了已经释放掉的存储
    cout << cs;
    if(strlen(cs=(s2+s3).c_str()) < 8
        && cs[0] == 'a') {
        // 在这里使用 cs
        // 同样无法保证正常工作
    }
}
```

```
void f(Shape& s, int x, int y)
{
    s.move(Point(x,y));
    // 构造一个Point并传递
    // 给Shape::move()
    // 这么临时用一下是可以的
}
```

10.4.11 对象的放置

- 按照默认方式，运算符new将在自由存储区创建对象，但是，也可以在其他地方分配对象

```
class X{
public:
    X(int);
    void* operator new(size_t,void* p){ return p; }
};

void* buf = reinterpret_cast<void*>(0xf00f);
X* p2 = new (buf)X; // 放置语法
// 在buf构造X时调用：operator new(sizeof(X), buf)
// 自由式放置可查阅15.6节
```

对象的放置示例

```
class Arena{
public:
    virtual void* alloc(size_t) = 0;
    virtual void free(void*) = 0;
    // …
};
void* operator new(size_t sz, Arena* a) {
    return a->alloc(sz);
}
```

```
void destroy(X* p, Arena* a){
    p->~X();    // 析构函数
    a->free(p); // 释放空间
}
```

使用时

```
extern Arena* Persistent;
extern Arena* Shared;
void g(int i) {
    X* p = new(Persistent) X(i); // 在持续性存储中分配X
    X* q = new(Shared) X(i);     // 在共享的存储中分配X
}
```

10.4.12 联合

- 命名联合的定义方式同struct，其中的各个成员将具有同样的地址
- 联合可以有成员函数，但是不能有静态成员
- 一般来说，编译器无法知道被使用的是联合的哪个成员，因此，联合就不能包含带构造函数或者析构函数的成员，因为无法保护其中的对象以防止破坏，也不可能保证在联合离开作用域时能调用正确的析构函数
- 最好是仅仅将联合用于底层代码，或者作为某个类的实现中的一部分，由这个类维护在联合中存储着什么信息

10.5 忠告

- [1]用类表示概念
- [2]只将public数据(struct)用在它实际上仅仅是数据，而且对于这些数据成员并不存在不变式的地方
- [3]一个具体类型属于最简单的类。如果适用的话，就应该尽可能使用具体类型，而不要采用更复杂的类，也不要用简单的数据结构
- [4]只将那些需要直接访问类的表示的函数作为成员函数
- [5]采用名字空间，使类与其协助函数之间的关系更明确
- [6]将那些不修改对象值的成员函数做成const成员函数
- [7]将那些需要访问类的表示，但无须针对特定对象调用的成员函数做成static成员函数

忠告

- [8]通过构造函数建立起类的不变式
- [9]如果构造函数申请某种资源，析构函数就应该释放这一资源
- [10]如果在一个类里有指针成员，它就需要有复制操作(包括复制构造函数和复制赋值)
- [11]如果在一个类里有引用成员，它就可能需要有复制操作(复制构造函数和复制赋值)
- [12]如果一个类需要复制操作或析构函数，它多半还需要有构造函数、析构函数、复制赋值和复制构造函数
- [13]在复制赋值里需要检查自我赋值

忠告

- [14]在写复制构造函数时，请小心地复制每个需要复制的元素(当心默认的初始式)
- [15]在向某个类中添加新成员时，一定要仔细检查，看是否存在需要更新的用户定义构造函数，以使它能够初始化新成员
- [16]在类声明中需要定义整型常量时，请使用枚举
- [17]在构造全局的和名字空间的对象时，应避免顺序依赖性
- [18]用第一次开关去缓和顺序依赖性问题
- [19]请记住，临时对象将在建立它们的那个完整表达式结束时销毁