USTC

Chapter 9
Technicalities: Classes, etc.

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 本章讨论关于用户自定义类型的技术细节，包括类和枚举
  - 类 Classes
    - 实现和接口
    - 构造函数
    - 成员函数
  - 枚举 Enumerations
  - 运算符重载 Operator overloading

类 Classes

- 思想：
  - “类”直接代表了程序设计的概念
    - 如果你能够把“它”看作为一个独立的实体，它就应该能表示为一个类或类的对象
    - 例如：vector, matrix, input stream, string, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
  - 类属于用户自定义类型，它指定了该类型对象的创建和使用方式

回顾一下类型的作用

  - 在 C++中 (多数现代语言亦是如此)，类是简化大型程序的核心组件
    - 当然，它对小程序也是很有用的
  - 类的概念最早源于早期语言 Simula67

成员和成员访问

- 一个类看起来是这个样子

```
class X { // this class' name is X
   // data members (they store information)
   // function members (they do things, using the information)
};
```

- 例如：

```
class X {
public:
   int m;    // data member
   int mf(int v) { int old = m; m=v; return old; }   // function member
};

X var;                          // var is a variable of type X
var.m = 7;                      // access var's data member m
int x = var.mf(9);   // call var's member function mf()
```

类

❖ 类是一种用户自定义类型

```
class X {         // this class' name is X
public:           // public members -- that's the interface to users
                  //          (accessible by all)
    // functions
    // types
    // data (often best kept private)
private:          // private members -- that's the implementation details
                  //          (accessible by members of this class only)
    // functions
    // types
    // data
};
```

类可以看作为：接口+实现

结构和类

- 类成员默认是私有的 (private)：

```
class X {
        int mf();
        // ...
};
```

  - 等价于

```
class X {
private:
        int mf();
        // ...
};
```

- 因此：

```
X x;                    // variable x of type X
int y = x.mf();         // error: mf is private (i.e., inaccessible)
```

结构和类

- 一个结构就是一个成员默认为公有属性的类：

```
struct X {
        int m;
        // …
};
```

  - 等价于

```
class X {
public:
        int m;
        // …
};
```

- 结构主要用于成员可以任意取值的数据结构

结构

```
// simplest Date (just data)   —— 本章示例
struct Date {
    int y, m, d;       // year, month, day
};

Date my_birthday;          // a Date variable (object)

my_birthday.y = 12;
my_birthday.m = 30;
my_birthday.d = 1950;      // oops! (no day 1950 in month 30)
                           // later in the program, we'll have a problem
```

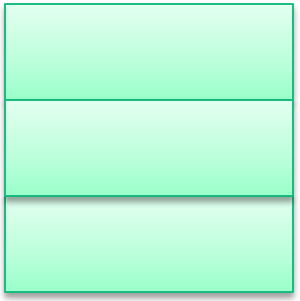
my_birthday:

Date:

y:

m:

d:

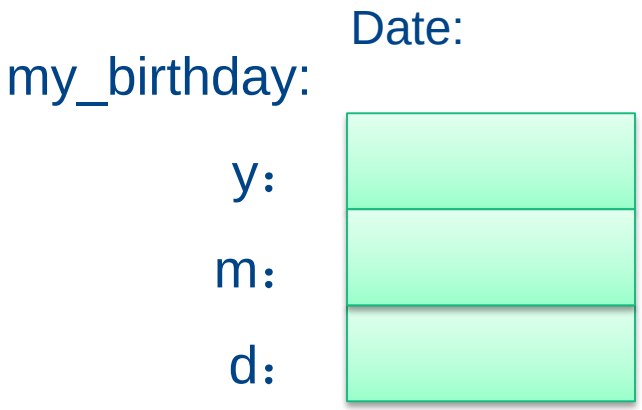


结构不能定义任何含义的不变式!

结构 — 辅助函数

```
// simple Date (with a few helper functions for convenience)
struct Date {
    int y,m,d;          // year, month, day
};

Date my_birthday;   // a Date variable (object)
```



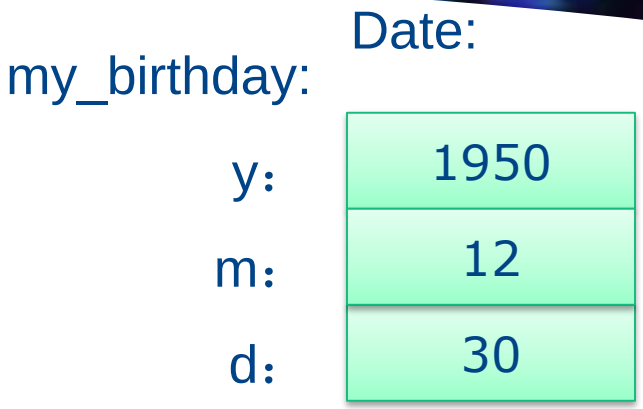
辅助函数，完成常见操作

```
// helper functions:
void init_day(Date& dd, int y, int m, int d); // check for valid date and initialize

void add_day(Date&, int n);     // increase the Date by n days
// …

init_day(my_birthday, 12, 30, 1950); // run time error: no day 1950 in month 30
```

结构 — 成员函数




```
// simple Date
//          guarantee initialization with constructor
//          provide some notational convenience
struct Date {
    int y,m,d;                  // year, month, day
    Date(int y, int m, int d);  // constructor: check for valid date and initialize
    void add_day(int n);        // increase the Date by n days
};

// …
Date my_birthday;                   // error: my_birthday not initialized
Date my_birthday(12, 30, 1950);     // oops! Runtime error
Date my_day(1950, 12, 30); // ok
my_day.add_day(2);                  // January 1, 1951
my_day.m = 14;                      // ouch! (now my_day is a bad date)
```

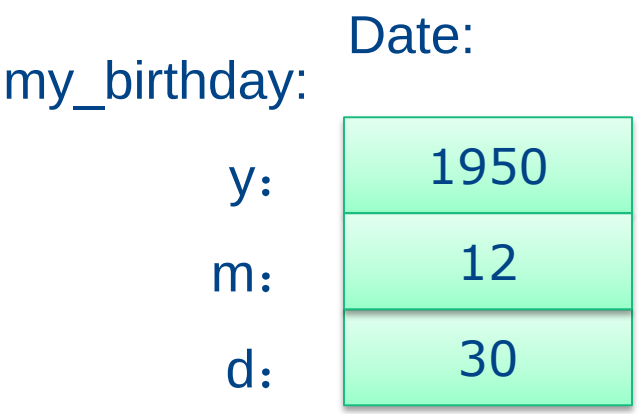
注意从辅助函数变换为成员函数的形式

演化为类

```
// simple Date (control access)
class Date {
    int y, m, d;        // year, month, day
public:
   // constructor: check for valid date and initialize
    Date(int y, int m, int d);

    // access functions:
    void add_day(int n);          // increase the Date by n days
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};

// …
Date my_birthday(1950, 12, 30);                 // ok
cout << my_birthday.month() << endl;            // we can read
my_birthday.m = 14;                             // error: Date::m is private
```



类

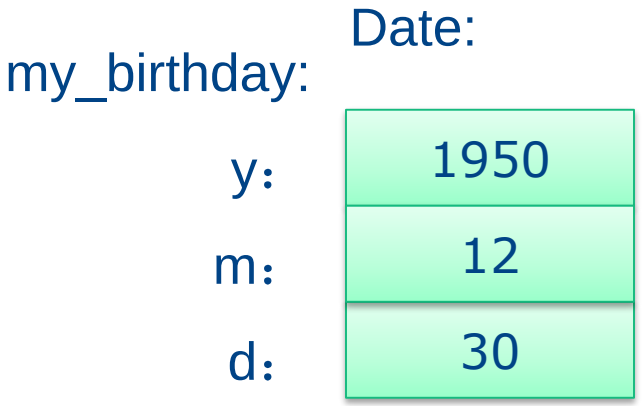
- 此处的“合法日期”是类“合法值思想”的一个重要特例
- 我们尽量仔细地设计类，以保证值的合法性
  - 或者,我们时常检查它的合法性，oops!
- 对合法值的约束规则称为“不变式”
  - Date的不变式(“Date必须正确的表示过去、现在或将来的一个实际日期”)通常难以精确满足
    - 如：闰年2月28日
- 如果我们不能构思一个好的不变式，就将它按普通数据结构进行处理
  - 这种情况下，使用结构(struct)
  - 尽力为你的类设计好的不变式
    - 如此，可将你从烦人的bug代码中解脱出来

类演化—实现

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    // constructor: check for valid date and initialize
    Date(int y, int m, int d);
    // increase the Date by n days
    void add_day(int n);
    int month();
    // ...
private:
    int y,m,d;          // year, month, day
};

Date::Date(int yy, int mm, int dd)                    // definition; note :: "member of"
    :y(yy), m(mm), d(dd) { /* ... */ };       // note: member initializers

void Date::add_day(int n) { /* ... */ };      // definition
```



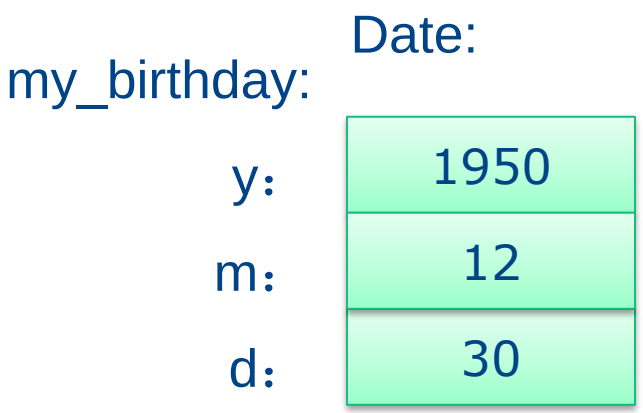
构造函数中的初始化

类演化— 实现

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    // constructor: check for valid date and initialize
    Date(int y, int m, int d);
    // increase the Date by n days
    void add_day(int n);
    int month();
    // …
private:
    int y,m,d;          // year, month, day
};

int month() {  return m; }        // error: forgot Date::
                                  // this month() will be seen as a global function
                                  //  not the member function, can't access members

int Date::season() { /* … */ }    // error: no member called season
```



类演化—不变式

```
// simple Date (what can we do in case of an invalid date?)
class Date {
public:
    class Invalid { };              // to be used as exception
    Date(int y, int m, int d);      // check for valid date and initialize
    // …
private:
    int y,m,d;                              // year, month, day
    bool check(int y, int m, int d);        // is (y,m,d) a valid date?
};

Date:: Date(int yy, int mm, int dd)
    : y(yy), m(mm), d(dd)                   // initialize data members
{
    if (!check(y,m,d)) throw Invalid();     // check for validity
}
```

构造函数不能返回值但可以抛出异常！

类

- 为什么要严格区分public/private呢?
- 为什么不让所有的属性都是public的呢?
  - 提供更加简洁的接口(对使用者)
    - 数据和复杂的实现函数是 private 的
  - 保持不变式约束
    - 只有有限的函数能够访问数据
  - 便于调试(对开发者)
    - 只有有限的函数能够访问数据
    - 称之为 “round up the usual suspects” 技术
  - 允许你改变它的描述(对维护者)
    - 你只需改动几个固定的函数即可
    - 你不知道究竟谁在使用public成员

接口与实现的分离

枚举 Enumerations

- enum (枚举)是一种非常简单的用户自定义类型，它指定了一个值的集合(枚举量， enumerators)

- 例如：

```
enum Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;
m = 7;                      // error: can't assign int to Month
int n = m;                  // ok: we can get the numeric value of a Month
Month mm = Month(7);        // convert int to Month (unchecked)
```

一个枚举量总是表示一个整数，而一个整数可能不是一个合法的枚举量！

枚举

- 是一些常量的简单列表：

```
enum { red, green };            // the enum { } doesn't define a scope
int a = red;                    // red is available here
enum { red, blue, purple };     // error: red defined twice
```

枚举量与枚举类型有着相同的作用域！

- 是一种常量列表类型

```
enum Color { red, green, blue, /* ... */ };
enum Month { jan, feb, mar, /* ... */ };

Month m1 = jan;
Month m2 = red;     // error red isn't a Month ——类型安全
Month m3 = 7;       // error 7 isn't a Month
int i = m1;         // ok: an enumerator is converted to its value, i==0
```

枚举—值

- 默认情况下

```
// the first enumerator  has the value  0,
// the next enumerator has the value "one plus the value of the
// enumerator before it"
enum { horse, pig, chicken };        // horse==0, pig==1, chicken==2
```

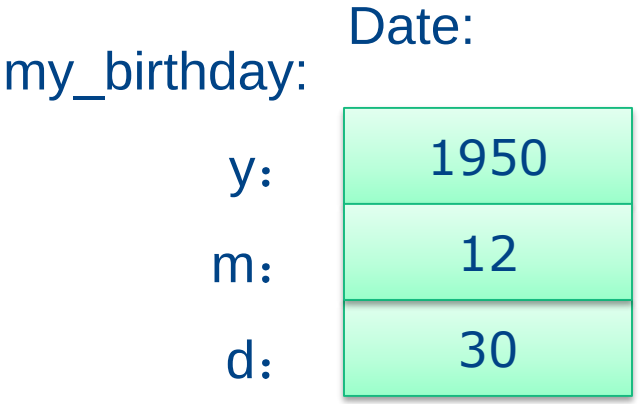
- 你能够控制基数值

```
enum { jan=1, feb, march /* ... */ };          // feb==2, march==3
enum stream_state { good=1, fail=2, bad=4, eof=8 };
int flags = fail+eof;                    // flags==10
stream_state s = flags;     // error: can't assign an int to a stream_state
stream_state s2 = stream_state(flags);      // explicit conversion (be careful!)
```

回到类

```
// simple Date (use Month type)
class Date {
public:
    enum Month {
        jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
    };
    Date(int y, Month m, int d);  // check for valid date and initialize
    // …
private:
    int y;          // year
    Month m;
    int d;          // day
};
```

通常将枚举类型定义于更窄的作用域中！




```
Date my_birthday(1950, 30, Date::dec);      // error: 2nd argument not a Month
Date my_birthday(1950, Date::dec, 30);      // ok
```

Const

```
class Date {
public:
    // …
    int day() const { return d; }        // const member: can't modify
    void add_day(int n);                 // non-const member: can modify
    // …
};
```

const成员函数不能更改数据成员

```
Date d(2000, Date::jan, 20);
const Date cd(2001, Date::feb, 21);

cout << d.day() << " – " << cd.day() << endl;   // ok
d.add_day(1);       // ok
cd.add_day(1);      // error: cd is a const
```

Const

```
//
Date d(2004, Date::jan, 7);          // a variable
const Date d2(2004, Date::feb, 28);  // a constant
d2 = d;              // error: d2 is const
d2.add(1);           // error d2 is const
d = d2;              // fine
d.add(1);            // fine

d2.f();    // should work if and only if f() doesn't modify d2
           // how do we achieve that? (say that's what we want, of course)
```

Const 成员函数

```
// Distinguish between functions that can modify (mutate) objects
// and those that cannot ("const member functions")
class Date {
public:
    // …
    int day() const;// get (a copy of) the day
    // …
    void add_day(int n);      // move the date n days forward
    // …
};

const Date dx(2008, Month::nov, 4);
int d = dx.day();     // fine
dx.add_day(4);        // error: can't modify constant (immutable) date
```

非const成员函数编译器认为会修改数据

const变量只能使用const成员函数

const从开发和使用上的两种含义？

类

- 好的接口是什么样的呢?
  - 最小化
    - 尽可能的小，这样才能更清晰
  - 完整
    - 保证够用，也不能太小
  - 类型安全
    - 小心让人迷惑的参数顺序
  - const纠正

用代码表述设计思想和使用约束!

类

基本操作

- 默认构造函数
  - 无定义情况下，编译器提供一个默认构造函数(什么都不做)
  - 如果定义了其他任何构造函数，则编译器不会再默认产生
- 拷贝构造函数
  - 默认产生的是拷贝每一个成员
- 拷贝复制操作
  - 默认产生的是拷贝每一个成员
- 析构函数
  - 默认情况下什么都不做

例如：

```
Date d;          // error: no default constructor
Date d2 = d;     // ok: copy initialized (copy the elements)
d = d2;          // ok copy assignment (copy the elements)
```

接口和“辅助函数”

- 保持类接口(public函数)的最小化
  - 简化理解
  - 简化调试
  - 简化维护

Why？如果类描述改变，只有直接访问描述的函数才需要重写

- 为了保持类接口的简洁，我们需要额外的“辅助函数”
  - 类外的非成员函数
  - e.g. == (equality) , != (inequality)
  - **next_weekday(), next_Sunday()**

辅助函数

```
Date next_Sunday(const Date& d)
{
    // access d using d.day(), d.month(), and d.year()
    // make new Date to return
}

Date next_weekday(const Date& d) { /* … */ }

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

bool operator!=(const Date& a, const Date& b) { return !(a==b); }
```

运算符重载

- 你可以在自己的类型(类、枚举)上定义几乎所有的C++运算符
  - 这通常称为“运算符重载”

```
enum Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month operator++(Month& m)              // prefix increment operator
{
    m = (m==dec) ? jan : Month(m+1);    // "wrap around"
    return m;
}

Month m = nov;
++m;        // m becomes dec
++m;        // m becomes jan
```

运算符重载

- 你只能定义已经存在的运算符而(不能定义新的运算符)
  - *e.g.*, + - * / % [] () ^ ! & < <= > >=
- 你定义的运算符必须与已有的C++语法一致(操作数个数)
  - *e.g.*, 不能定义一元的<=(小于等于)和二元的!(非)
- 一个重载的运算符至少有一个用户自定义类型的操作数
  - **int operator+(int,int);** // error: you can't overload built-in +
  - **Vector operator+(const Vector&, const Vector &);** // ok
- 建议(非语言规则)
  - 重载运算符应该保持其原有意义
  - + 应该是加法， * 应该是乘， [] 应该是访问， () 应该是调用，等等
- 建议(非语言规则)
  - 除非你真正确定重载运算符能大大改善代码，否则不要为你的类型定义运算符(非必要不要重载)!

Date类总结

```
Namespace Chrono{

class Date{
public:
    enum Month { … };

    class invalid { };

    Date(int y, Month m, int d);
    Date();
    // default copy operations are fine

    int day() const { return d; }
    Month month() const { return m; }
    int year() const { return y; }

    void add_day(int n);
    void add_month(int n);
    void add_year(int n);

private:
     int y;
    Month m;
    int d;
};

// true for valid day
bool is_date(int y, Date::Month m, int d);

// true if y is a leap year
bool leapyear(int y);

bool operator==(const Date& a, const Date& b);
bool operator!=(const Date& a, const Date& b);

ostream& operator<<(ostream& os, const Date& d);

istream& operator>>(istream& is, Date& dd);

} // Chrono
```

Next

- I/O 流
  - I/O基本概念
  - 文件操作
  - 错误处理