CISC 3142
Programming Paradigms in C++

Ch16 – Classes

Abstraction Mechanisms: Elements of OO Programming

(*Stroustrup – The C++ Programming Language, 4th Ed*)

Introduction

- C++ classes are for creating new types that can be used as built-in types
- Derived classes and templates allow expression of hierarchical and parametric relationships among classes
- Examples for user-defined types
  - Explosion for a video game
  - List<Paragraph> for a text-processing program
- Fundamental idea is to separate the details of the implementation from the description of how to use it (interface)
- This chapter focuses on simple “concrete” user-defined types

Class Basics

- Is a user-defined type
- There are data members and member functions
- Member functions define: initialization, copy, move, and cleanup
- Members are accessed via .(dot) for objects and -> for pointers
- A class is a namespace for its members
- The public members provide interface, and the private members provide implementation details
- A struct is a class where members are public by default

Class example

```
class X {
private:                          // the representation (implementation) is private
        int m;
public:                           // the user interface is public
        X(int i =0) :m{i} { }     // a constructor (initialize the data member m)
        int mf(int i) {           // a member function
            int old = m;
            m = i;                // set a new value
            return old;           // return the old value
        }
};

X var {7};                        // a variable of type X, initialized to 7

int user(X var, X∗ ptr) {
        int x = var.mf(7);        // access using . (dot)
        int y = ptr−>mf(9);       // access using -> (arrow)
        int z = var.m;            // error : cannot access private member
}
```

Member Functions

- Standalone functions

```
struct Date { // representation
    int d, m, y;
};
// initialize d
void init_date(Date& d, int, int, int);
// add n years to d
void add_year(Date& d, int n);
// add n months to d
void add_month(Date& d, int n);
// add n days to d
void add_day(Date& d, int n);
```

NO explicit connection between Date & functions

- Member functions

```
struct Date {
    int d, m, y;
    void init(int dd, int mm, int yy); // initialize
    void add_year(int n); // add n years
    void add_month(int n); // add n months
    void add_day(int n); // add n days
};
void Date::init(int dd, int mm, int yy) {
  d=dd; m=mm; y=yy; }
```

- Member functions can be invoked only by a variable of the Date type
- See that the Date object is implied – a class member function "knows" for which object it was invoked

Default Copying

- By default, a class object can be initialized with a copy of an object of its class

```
Date d1 = my_birthday; // initialization by copy
Date d2 {my_birthday}; // initialization by copy
```

- Such copy is a copy of each member (or memberwise copy)
- Similarly, class objects can be copied by assignment

```
void f(Date& d) {
    d = my_birthday;
}
```

- Again, the default semantics is memberwise copy
- If default behavior is not the right choice, user must define the appropriate copy and assignment operations.

Access Control

```
class Date {
    int d, m, y;
public:
    void init(int dd, int mm, int yy); // initialize
    void add_year(int n);              // add n years
    void add_month(int n);             // add n months
    void add_day(int n);               // add n days
};
```

- Default access in a class is *private*, The second, *public*, part constitutes the public interface to objects of the class
- Nonmember functions are barred from using private members, with the following benefits
  - Any error causing a Date to take on an illegal value must be caused by code in a member function – localization
  - A change in representation only need revision of member functions – user code need not be rewritten (just recompiling)
  - One only needs to study definitions of member function to learn on how to use a class
  - Focusing on the design of a good interface leads to better code, as debugging time is saved

Class definition is also known as declaration

- A class definition can be replicated in different source files using #include without violating the one-definition rule
- It’s not a requirement to declare data first in a class. Often it makes sense to place them last to emphasize the public functions as interface

```
class Date {
public:
    Date(int dd, int mm, int yy);
    void add_year(int n); // add n years
private:
    int d, m, y;
};
```

- Access specifiers can be used many times in a single class declaration, though a bit messy

Constructors

- When a class has a constructor, all objects of that class will be initialized by a constructor call

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);   // abbreviated form
Date my_birthday;        //error : initializer missing
Date release1_0(10,12);  // error : third argument missing
```

- Or using the {}-initializer notation → preferred

```
Date today = Date {23,6,1983};
Date xmas {25,12,1990}; // abbreviated form
Date release1_0 {10,12}; // error : third argument missing
```

Multiple Constructors

```
class Date {
        int d, m, y;
public:
        // ...
        Date(int, int, int);     // 1) day, month, year
        Date(int, int);          // 2) day, month, today's year
        Date(int);               // 3) day, today's month and year
        Date();                  // 4) default Date: today
        Date(const char∗);       // 5) date in string representation
};
```

- Constructors follow the same overloading rules as ordinary functions
- Examples of calling constructors:

```
Date today {4};                 // 3) 4, today.m, today.y
Date july4 {"July 4, 1983"};    // 5)
Date guy {5,11};                // 2) 5, November, today.y
Date now;                       // 4) default initialized as today
Date start {};                  // 4) default initialized as today
```

Default Arguments for Constructor

```
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; // what is it?
	m = mm ? mm : today.m;
	y = yy ? yy : today.y;
	// check that this Date is valid
}
```

- In this case, it’s common to pick an initial value that’s not legal (like 0, which delegating a default value to be specified)

```
class Date {
	int d, m, y;
public:
	Date(int dd =today.d, int mm
	=today.m, int yy =today.y);
	// ...
};
Date::Date(int dd, int mm, int yy) {
	// check that this Date is valid
}
```

- Alternatively, pick the default arguments directly as valid default values
- Both require building values (today) into Date’s interface (global?), which is undesirable

explicit Constructors

- By default, a constructor with a single argument acts as an implicit conversion from its argument type to its type

```
complex<double> d {1}; // d=={1,0} (§5.6.2)
```

- While this is useful for some cases, it could be a source of confusion and error

```
Date d = 15; // obscure, likely used as the date, instead of month or year
```

- We can specify that a constructor is not used as an implicit conversion (to be explicit)

```
class Date {
    int d, m, y;
public:
    explicit Date(int dd =0, int mm =0, int yy =0);
    // ...
};
Date d1 {15};           // OK: considered explicit
Date d2 = Date{15}; // OK: explicit
Date d3 = {15}; // error : = initialization does not do implicit conversions
Date d4 = 15; // error : = can't do implicit conversions
```

```
void my_fct(Date d);

void f()
{
    my_fct(15); // error : argument passing does not do implicit conversions
    my_fct({15}); // error : argument passing does not do implicit conversions
    my_fct(Date{15}); // OK: explicit
    // ...
}
```

Immutability

- Objects can be constant. Systematic use of immutable objects leads to
  - more comprehensible code
  - more errors being found early
  - sometimes improved performance
- For freestanding functions that operate on const objects of user-defined type
  - We simply use const T& as arguments
- For classes, we define member functions that work on const objects by specifying const after their argument list

Constant Member Functions

```
class Date {
    int d, m, y;
public:
    int day() const { return d; }
    int month() const { return m; }
    int year() const;
    void add_year(int n); // add n years
    // ...
};
```

- The const suffix is required again if the member function is defined outside its class

```
int Date::year()  { // error : const missing in member function type
    return y;
}
```

Self-Reference

- The benefit of returning a self reference:

```
void f(Date& d) {
    // ...
    d.add_day(1).add_month(1).add_year(1);
    // chained operation
    // ...
}
```

- Just need to declare the member functions to return a reference to Date:

```
class Date {
    // ...
    Date& add_year(int n); // add n years
    Date& add_month(int n); // add n months
    Date& add_day(int n); // add n days
};
```

- Example of add_year()

```
Date& Date::add_year(int n) {
  if (d==29 && m==2 && !leapyear(y+n)) {
    // beware of February 29
    d = 1;
    m = 3; // March 1st
  }
  y += n;
  return *this; // this is a pointer
} // note *pointer could be a reference
```

Static Members

- How do we implement the default value (say today) for Date?
- A global variable today would make the Date class too dependent on global context. The solution is a static member:

```
class Date {
   int d, m, y;
   static Date default_date; // can you include a non-static Date here?
public:
   Date(int dd =0, int mm =0, int yy =0);
   // ...
   static void set_default(int dd, int mm, int yy); // set default_date to Date(dd,mm,yy)
};
```

- Now the Date constructor can use it without globals

```
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;
      // ... check that the Date is valid ...
}
```

Note, you must initialize the static member outside the class like this:

```
Date Date::default_date {16,12,1770};
// definition of Date::default_date
```

Here, the keyword static is not repeated

Member Types

```
template<typename T>
class Tree {
    using value_type = T;                // member alias
    enum Policy { rb, splay, treeps }; // member enum
    class Node {          // member class, or nested class
        Node∗ right;
        Node∗ left;
        value_type value; // same as: T value
    public:
        void f(Tree∗);
    };
    Node∗ top;
public:
    void g(const T&);
    // ...
};
```

- A member class can refer to types and members (even private) of its enclosing class
- But it has no notion of a current object of the enclosing class

```
template<typename T>
void Tree::Node::f(Tree∗ p) {
    top = right; // error: no object of type Tree
                    specified
    p−>top = right; // OK
}
```

- A class doesn't have any special rights to the members of its nested class

```
template<typename T>
void Tree::g(Tree::Node∗ p) {
    value_type val = right−>value; //error: no object
                                      of type Tree::Node
    value_type v = p−>right−>value; // error:
                                      Node::right is private
    p−>f(this); //OK, f() is public
}
```

Concrete Classes (Example – a better Date)

```
namespace Chrono {
    enum class Month { jan=1, feb, mar, apr, may, jun, jul, aug,
                                        sep, oct, nov, dec };
    class Date {
    public: // public interface:
        class Bad_date { }; // exception class
        explicit Date(int dd ={}, Month mm ={}, int yy ={});
        // {} means ‘‘pick a default’’, here would be 0
        // nonmodifying functions for examining the Date:
        int day() const;
        Month month() const;
        int year() const;
        string string_rep() const; // string representation
        void char_rep(char s[], int max) const; // C-style string
                                                    representation
        // (modifying) functions for changing the Date:
        Date& add_year(int n); // add n years
        Date& add_month(int n); // add n months
        Date& add_day(int n); // add n days
    private:
        bool is_valid(); // check if this Date represents a date
        int d, m, y; // representation
    }; // end of class Date

    bool is_date(int d, Month m, int y); // true for valid date
    bool is_leapyear(int y); // true if y is a leap year

    bool operator==(const Date& a, const Date& b);
    bool operator!=(const Date& a, const Date& b);
    const Date& default_date(); // returns the default date
                                    not implemented yet
    ostream& operator<<(ostream& os, const Date& d); // print
    d to os
    istream& operator>>(istream& is, Date& d); // read Date
    from is into d
} // Chrono
```

Operations typical for a user-defined type

1. A constructor specifying how objects/variables of the type are to be initialized
2. A set of **const** functions allowing a user to examine a **Date**.
3. A set of functions allowing the user to modify **Date**s without actually having to know the details of the representation.
4. **Implicitly defined operations** that allow **Date**s to be freely copied
5. A class, **Bad_date**, to be used for reporting errors as exceptions.
6. A set of useful helper functions. The helper functions are not members and have no direct access to the representation of a **Date**, but they are identified as related by the use of the namespace **Chrono**.

Member Functions

- Constructor

```
Date::Date(int dd, Month mm, int yy)
        :d{dd}, m{mm}, y{yy} {
    if (y == 0) y = default_date().year();
    if (m == Month{}) m = default_date().month();
    if (d == 0) d = default_date().day();
    if (!is_valid()) throw Bad_date();
}
```

- is_date() (checking the d,m,y tuple) is potentially different from is_valid() (checking if a date is too old, i.e. more restricting)
- Trivial member functions:

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

Member Functions (cont')

- Non-trivial ones

```
Date& Date::add_month(int n) {
   if (n==0) return *this;
   if (n>0) {
      int delta_y = n/12; // number of whole years
      int mm = static_cast<int>(m)+n%12; // number of months ahead
      if (12 < mm) { // note: dec is represented by 12
         ++delta_y;
         mm -= 12;
      }
      // ... handle the cases where the month mm doesn't have day d ...
      // i.e. leapyear (so adding a month to 1/29 results in a valid date)
      y += delta_y;
      m = static_cast<Month>(mm);
      return *this;
   }
   // ... handle negative n ...
   return *this;
}
```

- This looks a bit messy, why?
- That's due to the d,m,y representation being inconvenient to computer as it is to us
- Better representation is simply to use the number of days since the Epoch (1/1/1970)

Helper Functions

- Helper functions are functions associated with a class that
  - need not be defined in the class, otherwise would make the class too complicated
  - don't have direct access to the representation

```
int diff(Date a, Date b); // number of days in the range [a,b) or [b,a)
bool is_leapyear(int y);
bool is_date(int d, Month m, int y);
const Date& default_date();
Date next_weekday(Date d);
Date next_saturday(Date d);
```

- In old C++, these functions are declared in the same place as the class declaration of Date (say Date.h)
- Alternatively, we can make the association explicit by enclosing the class and its helper function in a namespace. The definition should be put in a separate .cpp file

```
namespace Chrono { // facilities for dealing with time, likely more than just Date
    class Date { /* … */};
    int diff(Date a, Date b);
    bool is_leapyear(int y);
    // …
}
```

Overloaded Operators

- Overloaded operators enable conventional notation

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

- Other obvious choices

```
bool operator!=(Date, Date); // inequality
bool operator<(Date, Date);  // less than
bool operator>(Date, Date);  // greater than
// …
Date& operator++(Date& d) { return d.add_day(1); }           // increase Date by one day
Date& operator−−(Date& d) { return d.add_day(−1); }          // decrease Date by one day
Date& operator+=(Date& d, int n) { return d.add_day(n); }    // add n days
Date& operator−=(Date& d, int n) { return d.add_day(−n); }   // subtract n days
Date operator+(Date d, int n) { return d+=n; }               // add n days
Date operator−(Date d, int n) { return d+=n; }               // subtract n days
ostream& operator<<(ostream&, Date& d);                      // output d
istream& operator>>(istream&, Date& d);                      // read into d
```

Chapter-end Advice

[1] Represent concepts as classes; §16.1.

[2] Separate the interface of a class from its implementation; §16.1.

[3] Use public data (**struct**s) only when it really is just data and no invariant is meaningful for the data members; §16.2.4.

[4] Define a constructor to handle initialization of objects; §16.2.5.

[5] By default declare single-argument constructors **explicit**; §16.2.6.

[6] Declare a member function that does not modify the state of its object **const**; §16.2.9.

[7] A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures; §16.3.

[8] Make a function a member only if it needs direct access to the representation of a class; §16.3.2.

[9] Use a namespace to make the association between a class and its helper functions explicit; §16.3.2.

[10] Make a member function that doesn’t modify the value of its object a **const** member function; §16.2.9.1.

[11] Make a function that needs access to the representation of a class but needn’t be called for a specific object a **static** member function; §16.2.12.