Overview

• Classes
  • Implementation and interface
  • Constructors
  • Member functions

• Enumerations

• Operator overloading
Classes

• The idea:
  • A class directly represents a concept in a program
    • If you can think of “it” as a separate entity, it is plausible that it could be a class or an object of a class
    • Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
  • A class is a (user-defined) type that specifies how objects of its type can be created and used
  • In C++ (as in most modern languages), a class is the key building block for large programs
    • And very useful for small ones also
  • The concept was originally introduced in Simula67
Members and Member Access

• One way of looking at a class:
  ```cpp
class X { // this class’ name is X
    // data members (they store information)
    // function members (they do things, using the information)
}
```

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

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()
```
Classes

• A class X is a user-defined type

```cpp
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
};
```
Struct and class

• Class members are private by default:

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

• Means

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

• So

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

- A struct is a class where members are public by default:
  ```cpp
  struct X {
      int m;
      // ...
  };
  ```

- Means
  ```cpp
  class X {
      public:
      int m;
      // ...
  };
  ```

- **structs** are primarily used for data structures where the members can take any value
```c
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
```
// 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)

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

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

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

// simple Date
// guarantee initialization with constructor
// provide some notational convenience

```c
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
};
```

// ...

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

    // 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
Classes

• The notion of a “valid Date” is an important special case of the idea of a valid value

• We try to design our types so that values are guaranteed to be valid
  • Or we have to check for validity all the time

• A rule for what constitutes a valid value is called an “invariant”
  • The invariant for Date (“Date must represent a date in the past, present, or future”) is unusually hard to state precisely
    • Remember February 28, leap years, etc.

• If we can’t think of a good invariant, we are probably dealing with plain data
  • If so, use a struct
  • Try hard to think of good invariants for your classes
    • that saves you from poor buggy code
// simple Date (some people prefer implementation details last)
class Date {
public:
    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
    int month();
    // …
private:
    int y, m, d;  // year, month, day
};

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

void Date::add_day(int n) { /* … */ };  // definition
// simple Date (some people prefer implementation details last)
class Date {
public:
    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
    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
Classes

// 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
}
Classes

• Why bother with the public/private distinction?

• Why not make everything public?
  • To provide a clean interface
  • Data and messy functions can be made private
  • To maintain an invariant
  • Only a fixed set of functions can access the data
  • To ease debugging
  • Only a fixed set of functions can access the data
  • (known as the “round up the usual suspects” technique)
  • To allow a change of representation
  • You need only to change a fixed set of functions
  • You don’t really know who is using a public member
Enumerations

- An enum (enumeration) is a very simple user-defined type, specifying its set of values (its enumerators)
- For example:

```java
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)
```
Enumerations

- Simple list of constants:
  ```cpp
  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
  ```

- Type with list of constants
  ```cpp
  enum Color { red, green, blue, /* … */ };  
  enum Month { jan, feb, mar, /* … */ };  
  ```

  ```cpp
  Month m1 = jan;
  Month m2 = red;  // error red isn’t a Month
  Month m3 = 7;    // error 7 isn’t a Month
  int i = m1;      // ok: an enumerator is converted
                   // to its value, i==0
  ```
Enumerations – Values

• By default

```c
// 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
```

• You can control numbering

```c
enum { jan=1, feb, march /* … */ }; // feb==2, march==3
enum stream_state { good=1, fail=2, bad=4, eof=8 };
```

```c
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!)
```
Classes

// 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
};

Date my_birthday(1950, 30, Date::dec); // error: 2nd argument not a
    // Month
Date my_birthday(1950, Date::dec, 30); // ok
/**
 * Date d(2004, Date::jan, 7);       // a variable
 * Date const 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 Member Functions

// 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
    // ...
};

Date const dx(2008, Month::nov, 4);
int d = dx.day(); // fine
dx.add_day(4);    // error: can’t modify constant
                 // (immutable) date
class Date {
public:
    // ...
    int day() const { return d; } // const member: can’t modify
    void add_day(int n); // non-const member: can modify
    // ...
};

Date d(2000, Date::jan, 20);
Date const 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
Classes

• What makes a good interface?
  • Minimal
    • As small as possible
  • Complete
    • And no smaller
  • Type safe
    • Beware of confusing argument orders
  • Const correct
Classes

• Essential operations
  • Default constructor (defaults to: nothing)
    • No default if any other constructor is declared
  • Copy constructor (defaults to: copy the members)
  • Copy assignment (defaults to: copy the members)
  • Destructor (defaults to: nothing)

• For example

```
Date d; // error: no default constructor
Date d2 = d; // ok: copy initialized (copy the elements)
d = d2; // ok copy assignment (copy the elements)
```
Interfaces and “helper functions”

• Keep a class interface (the set of public functions) minimal
  • Simplifies understanding
  • Simplifies debugging
  • Simplifies maintenance

• When we keep the class interface simple and minimal, we need extra “helper functions” outside the class (non-member functions)
  • E.g. == (equality), != (inequality)
  • next_weekday(), next_Sunday()
Helper Functions

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

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

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

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

- You can define almost all C++ operators for a class or enumeration operands
- that’s often called “operator overloading”

```cpp
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
```
Operator Overloading

• You can define only existing operators
  • E.g., + - * / % [ ] ( ) ^ ! & < <= > >=

• You can define operators only with their conventional number of operands
  • E.g., no unary <= (less than or equal) and no binary ! (not)

• An overloaded operator must have at least one user-defined type as operand
  
  ```
  int operator+(int, int); // error: you can’t overload built-in +
  Vector operator+(Vector const &, Vector const &); // ok
  ```

• Advice (not language rule):
  • Overload operators only with their conventional meaning
  • + should be addition, * be multiplication, [ ] be access, ( ) be call, etc.

• Advice (not language rule):
  • Don’t overload unless you really have to