SFU CMPT-212 2008-1 Topic: Classes

Ján Maňuch

E-mail: jmanuch@sfu.ca

Friday 15th February, 2008

Encapsulation

- Using global variables can save you passing data to functions — they can easily access all data they need. However, it makes program unreadable and *unreliable* — data can be modified in unwanted manner.
- It's recommended to *protect* data from unnecessary access, i.e., pass to functions only the necessary data, and avoid using of global variables.
- In fact OOP dictates to completely *hide data* and provide only *interface* (functions) to access and modify the data (the user of the interface doesn't need to know how the data are actually stored in memory).
- Hiding implementation details and providing only the interface is called *encapsulation*, one of the main principles of OOP design.

C solution: header files

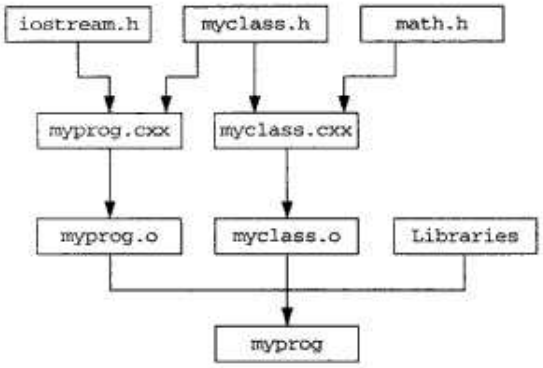
The code is split to several units where each unit groups related functions (e.g., `iostream` groups input/output functions together). Each unit has a header and a source code file. Typically,

- a *header file* — contains all declarations: function prototypes, constants, structure, template and class declarations;
- to each header file is assigned a *source code file* containing the definitions of the functions declared in the header file;
- the source code file could contain some global variables used for implementation of the function in it; and those are in some sense protected (only accessible in this source file)
- there is also a *source code file* containing the `main` function using the functions declared in header file(s).

Example: coordin.h, coordin.cpp, main.cpp [Prata]

Separate compilation

- *compiler* generates object `.o` files — only needs function prototypes to use the functions from other header files
- *linker* generates final executable file needs function definitions — each function is in the final executable file exactly once (unless `inline`)



Separate object code is generated for each source file. All object files generated during *compilation* and library files are joined together into the final executable during *linking*.

Question: why is it useful to split the code to several files?

Preprocessor

- *file inclusion* (`#include`)

  Examples:

```
1 #include <iostream>  // look in system directories
2 #include "coordin.h" // look in the local directory
```

- *macro definitions* (`#define`)

  Examples:

```
1   #define PI 3.14 // replace every occurrence of the
2     // string "PI" with the string "3.14" in the source
3     // file before compilations starts
4   #define COORDIN_H_ // defines "COORDIN_H_"
```

  macro definition can use arguments:

```
1 #define SWAP(a,b,type) { type temp=a; a=b; b=temp; }
```

  Example: swap6.cpp

- *conditional statements* (`#ifdef ...` or `#if defined ...`, `#ifndef ...` or `#if not defined ...`)

  Example:

```
1   #ifndef COORDIN_H_
2     ... code ...
3   #endif
```

Classes and objects

C solution to encapsulation is not sufficient.

Example. I want to use two stacks in my program. Either I would have to have two header/source code files, one for each stack (repeating the code), or one stack can (unintentionally) access data of the other.

C++ provides a better solution: ***classes***:

- represent abstraction of real objects
- *class* is a new user-defined type, and *objects* are variables of that type
- a **type** is described by two things:
  - how much memory is needed to store it
  - what operations can be performed on a variable of that type — collection of such operations is called *interface*

  Note. usually, we manipulate with data through the operations of the interface, not directly with bits representing the data in memory

Class specification

A class *specification* has two parts:

- *a class declaration* — describes data (data members) and interface (prototypes of member functions (also called *methods*)) [in a header file]
- *the class method definitions* — implementation of member functions [in the corresponding source code file]

Example of a *class declaration*:

```
1   class Date {
2    private:
3     int day, month, year;  // data members
4     bool incr_day();
5    public:
6     void set(int, int, int);
7     void get(int &, int &, int &);
8     void next();
9     void print();
10  };
```

- `Date` is a new type

```
1       Date d;
```

  declares a variable `d` of type `Date`, usually called *object*

- keywords `private` and `public` (and also `protected` discussed later in the course) describe *access control* for class members:
  - `private` *members* (data and methods) can be accessed only by methods of the class

```
1 d.day=10;      // compiler error
2 d.incr_day(); // compiler error
```

  - `public` *members* (data and methods) can be accessed from outside of the class

```
1 d.set(5,10,2005); // ok
```

- *Example:*
  `cout` is an object of class `ostream`
  `ostream` contains `public` methods `get()` and `getline()`, hence we can call them in the program using the membership operator ("`.`"):

```
1 cout.get();
```

Encapsulation with classes

- *data* and auxiliary functions are declared `private`: *data hiding*
- *interface* is declared `public`: hiding implementation details from the user of the class

client / server model:

- the class is a server
- the user of the class is a client — uses the server through a public interface
- the server and the client can modify (improve) their code independently (as long as they both follow the agreed interface)

Class methods definitions

- to identify that a function `f` belongs to class `X`, we have to use *the scope operator* `::`

```
1  void X::f(int a) {
2    // code
3  }
```

Example:

```
1    void Date::set(int d, int m, int y) {
2      day = d;
3      month = m;
4      year = y;
5    }
```

- members of class have a *class scope*;
  private members (`day`, `month` and `year`) can be accessed only in member functions of the class (`Date::set()`) as they are in the same (class) scope!

Complete example: date1.cpp

Using a class

```
1    Date d;
```

- creates an object `d` having its own copy of data members of its class (similarly as variables of type structure)
- all objects of the same class share one copy of each method
- the *membership* operator "`.`" can be used to access data members and methods

```
1    d.set(5,2,2004);
```

  - using a method `set()` on `d`
  - will call function `Date::set()` using data members of `d`

Example: date1.cpp

Note. A class (e.g. `Date`) can be used in a similar way as any built-in type:

- `Date today;`
- `Date *p = new Date;`
- `void f(Date &d);`
- `Date d, today;`
  `d = today;` — *memberwise assignment*
- `cout`, `cin` can be taught to print and read an object of our class

Sample questions

- Why is it recommended to prefer local declarations of variables to global declarations?
- What are the advantages of splitting C++ program to several files?
- What are the value of `i` and `j` after performing the following code:

```
1 #define SWAP(a,b,type) { type tmp=a; a=b; b=tmp; }
2 int i=1,j=2;
3 SWAP(++i,j,int)
```

- What is encapsulation? How is it useful? How it can be achieved for classes?
- What is the difference between private and public data member of a class?

Object initialization — Constructor

Examples of initializing:

- `int a=3;`
- `long b[4]={2L, 3L, 5L, -1L};`
- `struct Point { int x,y; } p={1,1};`

Question: How to initialize an object?

```
1 Date d={7,10,2005}; // compiler error:
2                     // accessing private members
```

- *Answer:* use **constructor** to initialize an object

```
1  class X {
2   public:
3    X(argument_list1);
4    X(argument_list2);
5  };
```

Example:

```
1  class Date {
2    int day, month, year;
3   public:
4    Date(int d, int m, int y); // constructor declaration
5    // ...
6  };
7
8  Date::Date(int d, int m, int y)
9  {
10   day = d; month = m; year = y;
11 } // constructor definition
```

Calling a constructor (initializing an object)

- explicitly:

```
1    Date d = Date(5,2,2004);
```

- implicitly:

```
1    Date d(5,2,2004);
```

- dynamic memory allocation:

```
1    Date *pdate = new Date(5,2,2004);
```

  (allocates space for *one* object of type `Date` and initializes its values using the constructor `Date(int,int,int)`)

Remark: Each time an object is declared/allocated, the constructor is automatically called!

Question: Which constructor is called when an object is declared/allocated without calling a constructor (`Date d;`)?

The default constructor (with no arguments)

- *if no constructor is defined,* the compiler automatically provides a *default constructor*:

```
1   X::X() {} // default constructor for class X
```

  having no arguments and doing nothing

- calling the default constructor:
  - explicitly:

```
1       Date d = Date();
```

  - implicitly:

```
1       Date d;
2       Date d(); // compiler error!
```

  - dynamic memory allocation:

```
1       Date *pdate = new Date;
```

- if there is a user-defined constructor, the default constructor is *not* provided:

```
1 Date d; // tries to call the default constructor
2         //   - a compiler error
```

- it's a good idea to provide the default constructor

  Question: Why?

  – `Date d;`

  – to declare arrays of objects

```
1 Date d[10]; // compiler error:
2              // wants to call default constructor (10 times)
3 Date *p=new Date[10]; // compiler error again
```

- it's a good idea to make the default constructor to initialize data members to a meaningful default value

Example: date2.cpp

Remark: Constructor `Date(int d=7,int m=10,int y=2005);` provides also the default constructor.

Destructor

- when the object seize exists, the destructor is *automatically* called!

```
1 class X {
2  public:
3    ~X();
4 };
```

- no arguments! hence, there can be only *one* destructor
- if no defined, it's provided automatically by compiler:

```
1 X::~X() {} // do nothing
```

- *Question:* When is called?
  - *automatic* storage object: when the program exists the block
  - *static* storage object: when program terminates
  - *dynamic* storage object: when `delete` is called

Example: date3.cpp

Question: What is the use of the destructor?

- to clean-up after we finish using an object

Example:

```
1   class intArray {
2     int *data;
3     int size;
4    public:
5     intArray(int s);
6     ~intArray(); // destructor
7   };
8
9   intArray::intArray(int s) {
10    size = s;
11    data = new int[size];
12  }
13  intArray::~intArray() {
14    delete [] data;
15  };
```

const methods

Example:

```
1   const Date d = Date(5,2,2004);
2   d.set(1,1,1888); // compiler error
3   d.print();       // also a compiler error
```

Q: How to specify that the member function doesn't modify the object?

Example:

```
1   class Date {
2     ...
3     void print() const;
4   };
5   void Date::print() const
6   { ... }
```

Example: date4.cpp

Remark: All accessor methods should be defined as `const`!

The this pointer

- each class method is passed a hidden parameter

```
1    X * const this;
```

  a `const` pointer to the object on which the method was invoked

- *Example:*

```
1    class X {
2      int m;
3     public:
4      int value();
5    };
6
7    int X::value() { return m; } // is equivalent to
8    int X::value() { return this->m; }
```

Question: What for is the `this` pointer?

Example:

- `cin.get(char [],int)` function returns a reference to `cin` which allows chaining the calls:

```
1 cin.get(buffer,MAX_SIZE) >>ch;
```

- *Q.* Define a member function `later(Date &)` which returns a reference to the object representing more recent date:

```
1 class Date {
2   // ...
3   Date &later(Date &d);
4 };
```

  If the object on which the function `later()` was invoked is more recent, we can use `*this` to get the reference to this object.

```
1  Date &Date::later(Date &d)
2  {
3    if (year>d.year ||
4        year==d.year &&
5         (month>d.month ||
6          month==d.month && day>d.day))
7      return *this;
8    else
9      return d;
10 }
11
12 // example of use:
13 Date today;
14 Date thanksgiving(10,10,2005);
15 today.later(thanksgiving).print();
```

Class inline functions

- Member functions can be defined directly in the class declaration:

```
1 class Point {
2  private:
3   int x,y;
4  public:
5   Point(int ix,int iy) { x=ix; y=iy; }
6   void move(int mx,int my) { x+=mx; y+=my; }
7   int getX() const { return x; }
8   int getY() const { return y; }
9 };
```

- They are automatically declared as `inline` functions! (should be used only for functions with short code)

Static members of class

- declared with the keyword `static`
- can be both data members and member functions:
  1. *data members*: there is only *one* copy of a variable or a constant for the whole class

     Examples:

```
1    // in header file:
2    class X {
3     private:
4      static const int MAX=100; // only const integer types
5        // can be initialized directly in the class declaration
6      static int nobjects;
7    };
8
9    // in source code file:
10   int X::nobjects=0; // initialization
```

     Example: date5.cpp

2. *member functions*:
   - doesn't have access to any particular object $\Longrightarrow$ can only access static members of class
   - can be also called using the class name (not through the object) — of course, only if it declared in `public` section

   Example:

```
1   class X {
2     ...
3    public:
4     static int numObjects();
5   };
6   // in source code file:
7   int X::numObjects() { return nobjects; }
8   // in main file:
9   cout << X::numObjects();
```

   Example: date6.cpp

Sample questions

- Consider a class `Int` with a unary constructor:

```
1 class Int {
2   int value;
3  public:
4   Int(int v) { value=v; }
5 };
```

  Show 3 different ways of how to create an object of type `Int` with value 10. Give a *short* description for each of them. (There is another way, see the next Sample questions.)

- What is the default constructor? When is the default constructor provided for a class by the compiler? Why is it useful to define the default constructor?

- What is the output of the following program?

```
1  #include <iostream>
2  using namespace std;
3
4  class Int {
5    int value;
6   public:
7    Int(int v) { value=v; }
8    ~Int() { cout << value << endl; }
9  };
10
11 Int a(1);
12
13 int main() {
14   Int b(2);
15   Int *c = new Int(3);
16   {
17     Int d(4);
18   }
19   delete c;
20   return 0;
21 }
```

- What's wrong with the following code?

```
1  class Int {
2    int value;
3   public:
4    Int(int v) { value=v; }
5    int get() { return value; }
6  };
7
8  int main()
9  {
10   const Int i(10);
11   cout <<i.get();
12 }
```

- What is `this` pointer? Show an example when you need to use `this` pointer.
- How would you count the number of existing objects of a class `X`? How would you count the total number of all objects of type `X` declared/allocated?

User-defined type conversions

- *unary constructors* provide automatic conversions *to* the class type

  Example:

```
1 class Date {
2   // ...
3  public:
4   Date(int d=0, int m=0, int y=0); // constructor
5   // zero values will initialize to current date
6   // also provides a unary constructor Date(int)
7   // ...
8 };
```

- The following statements are equivalent:

```
1 Date d=10;        // implicit conversion
2 Date d=Date(10); // explicit conversion
3 Date d=(Date)10; // explicit conversion
```

  Example: date7.cpp

- implicit conversions are sometimes confusing; is there a way how to forbid them?

  "YES" — use the keyword `explicit` in declaration of the constructor!

```
1  class Int {
2    int i;
3   public:
4    explicit Int(int ii) { i=ii; } // unary constructor
5  };
6  ...
7  Int number=10; // compiler error:
8                 //   implicit conversion not allowed
9  Int number=Int(10);  // ok
10 Int number=(Int) 10; // ok
```

- two-step conversion:

```
1    Date d=12.33; // ok: 2 conversions
```

  - at most one user-defined conversion
  - works only if there is an unambiguous choice

```
1  class Int {
2    int i;
3   public:
4    Int(int ii) { i=ii; } // unary constructor
5  };
```

- when is implicit conversion used?
  - initialization: `Int x=10;`
  - assignment: `x=12;`
  - passing `int` to a function expecting `Int` (by value or `const` reference)

```
1  void f(const Int&);
2  ...
3  f(10); //ok
```

  - when returning `int` in function with return value of type `Int` (by value or `const` reference)

- how about converting *from* the class type to other type (basic type, or to the class type which declaration we cannot modify)?
  use *conversion functions*!

- *Example (converting to the class type):*

```
1   class Time {
2    private:
3     int hours,minutes;
4    public:
5     Time(int m=0,int h=0) {
6       minutes = m % 60;
7       hours   = h + m/ 60;
8     }
9   };
```

now, we can convert `int` (minutes) to `Time`:

```
1   Time t=123; // t={2,3}
```

Question: how to convert `Time` to `int`?

```
1  class Time {
2  ...
3   operator int(); // conversion function
4  ...
5  };
6  ...
7  Time::operator int() {
8    return 60*hours+minutes;
9  }
```

now, we can convert `Time` to `int`:

```
1    Time t(10,2);  // 2 hours, 10 minutes
2    int i=(int) t; // explicit conversion: i=130
3    int j=int(t);  // explicit conversion: j=130
4    int k=t;       // implicit conversion: k=130
```

syntax:

```
1    operator type_name();
```

Example: time1.cpp

Friend member functions

- have access to private members of the class
- are not member functions
- defined in the same scope as the class (not inside of the class)

Example:

```
1  class X {
2   private:
3    int a;
4    void modify() { a += 2; }
5   public:
6    X(int i) { a=i; }
7    friend void f(int,X &);
8  };
```

```
1  void f(int i, X &x) {
2    x.modify();   // we can access private member function
3    cout <<i*x.a; // we can access private data member
4  }
5
6  // in the program:
7  X x(3);
8  f(10,x);   // prints 50
9  x.f(10,x); // compiler error:
10            //   f is not a member function!
```

Remarks.

- `friend` functions does not violate *data hiding*:
- `friend` function to a class `X` can be only declared in the class `X`
- they are just different mechanism for expressing *class interface*

Sample questions

- Consider a class `Int` with a unary constructor:

```
1  class Int {
2    int value;
3   public:
4    Int(int v) { value=v; }
5  };
```

  Show 5 different ways of how to create an object of type `Int` with value 10. Give a *short* description for each of them.

- Consider a class `Point`:

```
1  class Point {
2    int x,y;
3   public:
4    Point() { x=0; y=0; }
5    Point(int nx, int ny) { x=nx; y=ny; }
6    int getX() const { return x; }
7    int getY() const { return y; }
8  };
```

Enhance the class with methods allowing conversions from an `int` value and to a `double` value, where:

1. an `int` value `v` is converted to a point `(v,0)`;
2. a point is converted to a `double` value representing its distance from the origin (that is from point `(0,0)`).

Example:

```
1 Point p1=1;
2 cout <<p1.getX()<<" "<<p1.getY()<<endl; // prints "1 0"
3 Point p2(1,1);
4 double d=p2;
5 cout d<<endl; // prints 1.41 (square root of 2)
```

Hint: You can use a `cmath` function with prototype `double sqrt(double);` returning square root of the argument.