CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 8 – Classes:

Access control, constructors, and destructors

Announcement

- In person on 2/15 in CLSSRM 116
- Quiz on 2/17 in during lecture
- Reading assignment
  - Ch. 14
  - http://www.cplusplus.com/doc/tutorial/inheritance/

Access Control: friends

- Private members of a class cannot be accessed outside of class
- Generic functions and classes can be declared to be a “friend” and gain access to private members
- Within the class, precede function declaration with keyword ***friend***

Access Control: friends

```
//: C05:Friend.cpp

struct X; // incomplete type specification (or forward
declaration)
          // needed for the definition of f

struct Y {
  void f(X*);
};

struct X { // Definition
 private:
  int i;
 public:
  void initialize();
  friend void g(X*, int); // Global friend
  friend void Y::f(X*);  // Struct member friend
  friend struct Z; // Entire struct is a friend
  friend void h(); // Another global friend
};
```

Access Control: friends

- A typical use of friend functions is to give access to low-level functions that perform special operations in a class:

```
class MyWindow
 {
    // a low-level OS function needs friend access so that it can control
    // key functionality of our window class: to signal when to draw
    friend void ::sysdraw ( MyWindow* );

   public:
    // send a window redraw request to the OS, ok to be public:
    void redraw();

   private :
    // the draw() function is private because it should only be called by
    // the OS (via sysdraw) when the drawing context is ready to be used:
    void draw ();
 };
```

Stash class with access control

```
//: C05:Stash.h
// Converted to use access control
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);

public: // here is the public interface, some coding styles will
        // prefer the interface to appear before data member declarations
  void initialize(int size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};

#endif
```

Hiding implementation from interface

- We may not to want to have our implementation visible to our client
  - Our competitors may be able to obtain it
  - For security reasons: encryption algorithm
  - To prevent others from “cracking” our program

Hiding implementation from interface

```
// Window.h:
class Window {
  struct Internal;  // Forward declaration only
  Internal* intwin; // Put in Internal all the many private data and methods
                    // Internal is only declared in the .cpp
                    // (ok since the size of a pointer is type independent)
public:
   void init ();
   int run ();
};
```

```
// Window.cpp:
struct Window::Internal {
     int i, a, b;
     void readEvents ();
     void wait ();
     ...
};

Window::init () { intwin = new Internal; ...  }
Window::run () { intwin->readEvents(); ... }
```

Wake up

- https://youtu.be/c5n6lnEineQ

Constructors

- A constructor is a special function to initialize objects.
  - Avoid undetermined results.
  - Executed at creation of object.
  - Cannot be called like any other function.
  - No return and no void.

```
Class CRectangle {
        int width, height;
    public:
        CRectangle (int ,int );   // constructor declaration
        int area () {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {    //definition
    width = a;
    height = b;
}
```

Overloading Constructors

- Constructors can be overloaded, like operators and functions.
  - Defined multiple times.
  - For different number of parameters or types.
  - The one called is the one with matching parameters.
  - A constructor without input parameter is called a default constructor.

Constructors

```
class X {
     int i;
   public:
     X()                // Default Constructor
     { i=0; }

     X(int n)           // Alternative constructor
     { i=n; }
   };


   void f() {
     X x1;        // Default constructor called
     X x2(3);     // Alternative constructor called
     ...
   }
```

Destructors

- A destructor is a special function to destroy objects.
  - Release dynamically allocated memory.
  - When an object is created with ***new***, destructor is called upon delete.
  - When an object is created ***locally*** within a function, destructor is called when function returns.

```
~CRectangle () {  // destructor definifion
    delete …;
    …
    delete …;
}
```

Destructors

```
class X {
      int i;
    public:
      X()                 // Default Constructor
      { i=0; }

      X(int n)            // Alternative constructor
      { i=n; }

     ~X() { ... };        // Destructor (only one can exist)
    };


    void f() {
      X x1;       // Default constructor called (x1.i is 0)
      X x2(3);    // "int constructor" called (x2.i is 3)
      ...
    }  // At the end of their scope, a and b destructors will
       // be automatically called!
```

Revisiting the Stash class

```
//: C06:Stash2.h
// With constructors & destructors

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size);   // Constructor takes care of initialization
  ~Stash();          // Destructor
  int add(void* element);
  void* fetch(int index);
  int count();
};
```

Revisiting the Stash class

```
// Constructor:
Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

// Destructor:
Stash::~Stash() {
  if(storage != 0) cout << "freeing storage" << endl;
  delete []storage;
}
```

Aggregate Initialization

- We can initialize an array of any primitive type with aggregate initialization:

```
int a[5] = { 1, 2, 3, 4, 5 }; // 1) each element goes to one "entry" of a

int b[6] = {0}; // 2) all the missing elements of b will be initialized to 0

int b[6];       // 3) none of the elements are initialized

int c[] = { 1, 2, 3, 4 }; // 4) this is also valid, the size of c
                             will be automatically counted from the
                             number of initialized values

sizeof(c)          // 5) will return the size (in Bytes)of the entire array c

size c             // 6) will return the number of entries in the array c
```

Aggregate Initialization

- We can initialize an array of classes with aggregate initialization:

```
struct X {
  int i; float f; char c;
};

X x1 = { 1, 2.2, 'c' }; // 1) initialization of one object
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} }; // 2) initialization of two objects,
                                            //    the 3rd object is
                                            //    initialized to 0


struct Y {
  float f; int i;
  Y(int a);
};

Y y1[] = { Y(1), Y(2), Y(3) }; // initialization using constructor
```