CSE 165/ENGR 140
Intro to Object Orient Program

Lecture 8 – Classes:
Access control, constructors, and destructors
Announcement

- Lab #3 tomorrow
  - Due date: 2/19 (Thursday) at 11:59PM.

- Lab policies
  - Results will be released on Thursdays
  - Have until the following Mondays to re-do
  - Try to attend lab sessions. Extra information given during labs is important (points docked from assignments because of missing this information will not be credited).
  - DO NOT COPY or TURN IN SOMEONE ELSE’S WORK.

- Quiz on 2/19 in during lecture

- Reading assignment
  - Ch. 14
Access control is defined with keywords:
- private: accessible only by original/base class
- protected: accessible by base and derived (inheritance) classes
- public: accessible by everyone

```cpp
struct A {
    float val; // in a struct, members are public by default
    private:   // but we may change the access for the next members
        int size; // private members can only be accessed by methods of A
};

class C {
    float val; // in a class members are private by default
    public:    // but we may change the access to public
        int size; // now this member is private
};
```
struct A {
    float val; // 1) in a struct members are **public by default**
    private:
        int size; // 2) private members can only be accessed by methods of A
    protected:
        float x; // 3) protected members are similar to private,
                 // but inherited classes are given full access to them
    public:
        void setSize () {}; // 4) resize represents an interface method to class A
    private:
        void freemem () {}; // 5) this method is used for internal operations only
    protected:
        void inflate () {}; // 6) internal method accessible by derived classes
};

int main() {
    A a;
    a.val = 0.1f; // ok
    a.setSize(); // ok
    a.size = 3; // compilation error (member inaccessible)
    a.inflate(); // compilation error
}
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:

```cpp
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();
};
```
Access Control: friend class

- Making a structure nested doesn’t automatically give it access to private members.
- 3 steps:
  - Declare (without defining) the nested structure.
  - Declare it as a friend.
  - Define the structure.
Access Control: friend class

//: C05:NestFriend.cpp

struct Holder {
    private:
        int a[sz];
    public:
        void initialize();
        struct Pointer;     // 1) forward declaration
    friend struct Pointer;   // 2) declare Pointer as friend

    struct Pointer {       // 3) define Pointer
        private:
            int* p;
        public:
            void initialize();
            ...
            int read();     // 4) need to access values of a
            void set(int i); // 5) need to access values of a
    };
};
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
We may not 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(); ... }
Constructors

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

```cpp
class CRectangle {
    int width, height;
    public:
        CRectangle(int a, int b); // 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.
class X {
    int i;
public:
    X() // Default Constructor
    { i=0; }

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

void f() {
    X a; // Default constructor called (a.i will have 0)
    X a(3); // Alternative constructor called (a.i will have 3)
    ...
}
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.

```cpp
~CRectangle () { // destructor definition
    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 a; // Default constructor called (a.i will have 0)
    X b(3); // "int constructor" called (b.i will have 3)
    ...
} // At the end of their scope, a and b destructors will
    // be automatically called!
//: 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;
}
We can initialize an array of any primitive type with aggregate initialization:

```c
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
sizeof c / sizeof *c  // 6) will return the number of entries in the array c
```
We can initialize an array of classes with aggregate initialization:

```c
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
```