

CSE 165/ENGR 140

Intro to Object Orient

Program

Lecture 6 – Data Abstraction



Announcement

- ▶ Reading assignment
 - Ch. 4
 - <http://www.cplusplus.com/doc/tutorial/classes/>

Data Abstraction

- ▶ What is the main point behind “data abstraction”?

Wikipedia says:

*“In computer science, **abstraction** is the process by which data and programs are defined with **a representation similar in form to its meaning** (semantics), while **hiding away the implementation details**. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. A system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the computer hardware where the program is run, while high-level layers deal with the business logic of the program.”*

Simplify data access and use:

Hide details and design appropriate manipulation interface

Object oriented concepts

▶ Encapsulation

- The ability to package data with functions
- Variables are encapsulated in a class/structure with member functions (methods)

▶ Implementation hiding

- Access control
- To prevent important data from being corrupted

▶ Interface

- It establishes what requests you can make for a particular object
- It is an abstraction of an object
- Tells what an object does without the details (i.e. header files)

Structures in C vs C++

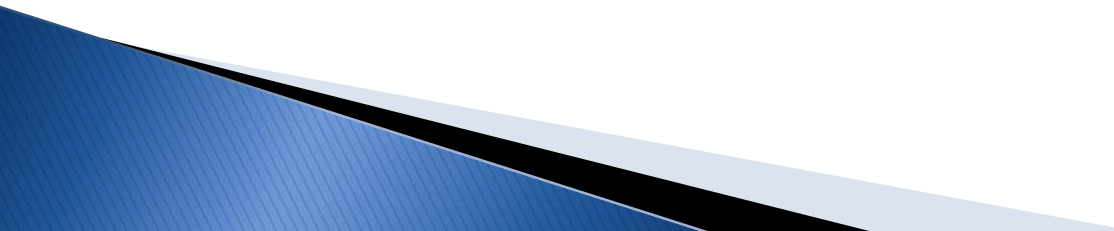
▶ C Structures

- Cannot have member functions inside structure
- Cannot directly initialize member variables

▶ C++ Structures

- Can have member functions (methods) in structure
- Can initialize member variables
- Almost the same as a class with one little difference:
 - Structures default to public visibility and classes to private
- **Convention is to use Structs for just data and classes for data and methods**

Libraries

- ▶ Code that someone else has written and packed together
 - ▶ We can utilize libraries to increase our productivity
 - ▶ Consists of a library file and header files
 - ▶ To be able to use libraries efficiently, we must understand how libraries work
- 

C-Like Stash library interface

```
// Header file for an array-like class

typedef struct CStashTag { // (recall the typedef is only needed in C)
    int size;           // Size (bytes) of each entry
    int quantity;       // Number of storage spaces (entries allocated)
    int next;           // Next empty space (equal to the number of elements)
    unsigned char* storage; // Dynamically allocated array of bytes
} CStash;

// Common C-like function naming style to avoid name clashes:

void  cstash_initialize ( CStash* s, int size );
void  cstash_cleanup   ( CStash* s );
int   cstash_add       ( CStash* s, const void* element );
void* cstash_fetch     ( CStash* s, int index );
int   cstash_count     ( CStash* s );
void  cstash_inflate   ( CStash* s, int increase );
```

C-Like Stash class

```
//: C04:CLib.cpp {0}
// Implementation of example C-like library
// Declare structure and functions:
#include "CLib.h"           // Include the class header file
#include <iostream>
#include <cassert>
using namespace std;

// Quantity of elements to add when increasing storage:
const int increment = 100;

void initialize ( CStash* s, int size ) {
    s->size = size;  //Unit size of element in bytes
    s->quantity = 0;
    s->storage = 0;
    s->next = 0;
}
```


C-Like Stash class

```
//: C04:CLib.cpp - continue
```

```
int add ( CStash* s, const void* element ) {  
    if ( s->next >= s->quantity ) // Not enough space left  
        inflate(s, increment); // Inflate the stash  
  
    // Copy element into storage, starting at next empty space:  
    int startBytes = s->next * s->size; // Locate next available position in  
                                       // storage  
    unsigned char* e = (unsigned char*)element; // Cast element from  
                                                // type void to unsigned char  
    for ( int i=0; i < s->size; i++ )  
        s->storage[startBytes + i] = e[i]; // Copy character by character  
  
    s->next++; // Update next available index  
    return(s->next - 1); // Index number of last entry  
}
```

C-Like Stash class

```
//: C04:CLib.cpp - continue
```

```
void inflate(CStash* s, int increase) {  
    assert(increase > 0); // Make sure expansion is positive  
  
    int newQuantity = s->quantity + increase;  
    int newBytes = newQuantity * s->size; // Total memory in bytes  
    int oldBytes = s->quantity * s->size; // Total memory in bytes  
  
    unsigned char* b = new unsigned char[newBytes]; // New array  
    for(int i = 0; i < oldBytes; i++)  
        b[i] = s->storage[i]; // Copy old to new  
  
    delete [] (s->storage); // Old storage  
  
    s->storage = b; // Point to new memory  
    s->quantity = newQuantity;  
}
```

C-Like Stash class

```
//: C04:CLib.cpp - continue
```

```
void* fetch(CStash* s, int index) {
    // Check index boundaries:
    assert(0 <= index);
    if(index >= s->next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(s->storage[index * s->size]);
}

int count(CStash* s) {
    return s->next; // Number of elements in CStash
}

void cleanup (CStash* s) {
    if ( s->storage!=0) {
        cout << "freeing storage" << endl;
        delete []s->storage;
    }
}
```

Dynamic Storage allocation

- ▶ Heap memory
 - Memory set aside by the program during runtime
- ▶ In C:
 - malloc, calloc, realloc, free
- ▶ In C++:
 - new, delete

Dynamic Storage allocation

- ▶ pointer = new type
- ▶ pointer = new type [number_of_elements]
- ▶ Examples:
 - `double* p_variable;`
 - `p_variable = new double;`

 - `int * a;`
 - `a = new int [5];`

 - `vehicle * p_vehicle;`
 - `p_vehicle = new vehicle;`

Dynamic Storage allocation

- ▶ delete pointer;
- ▶ delete [] pointer;
- ▶ Examples:
 - delete p_variable;
 - delete [] a;
 - delete p_vehicle;

Using the C-Like Stash class

```
//C04:CLibTest.cpp (simplified)
int main() {

    //1. Define variables at the beginning of the block, as in C:
    CStash stash;

    //2. Now remember to initialize our object:
    initialize ( &stash, sizeof(int) );

    //3. Now let's add some elements:
    for (int i = 0; i < 100; i++)
        add ( &stash, &i );

    //4. Now let's print the contents:
    for(int i = 0; i < count(&stash); i++)
        cout << * ((int*)fetch(&stash, i)) << endl; // Cast from void* to int*

    cleanup(&stash);
}
```

Using the C-Like Stash class

- ▶ Difficulties:
 - Manipulation of void pointers
 - Many type conversions needed
 - Long naming conventions: functions from different classes cannot have the same name
 - Explicit initialization and cleanup calls needed
 - Syntax long and sometimes not trivial
- ▶ Let's now re-write the same class in C++

C++ Stash class

```
//: C04:CppLib.h
// C-like library converted to C++

struct Stash {
    int size;           // Size of each space
    int quantity;       // Number of storage spaces
    int next;           // Next empty space
    unsigned char* storage; // Dynamically allocated array of bytes

    // Methods
    void initialize(int size);
    void cleanup();
    int add(const void* element);
    void* fetch(int index);
    int count();
    void inflate(int increase);
};
```

C++ Stash class

```
//: C04:CppLib.cpp {0}
// C library converted to C++
// Declare structure and functions:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add when increasing storage:
const int increment = 100;

void Stash::initialize(int sz) {
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}

void Stash::cleanup() {
    // There is no need to test if(storage!=0),
    // operator delete will already make the test.
    delete []storage;
}
```

C++ Stash class

```
int Stash::add(const void* element) {
    if(next >= quantity) // Not enough space left
        inflate(increment);

    // Copy element into storage, starting at next empty space:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];

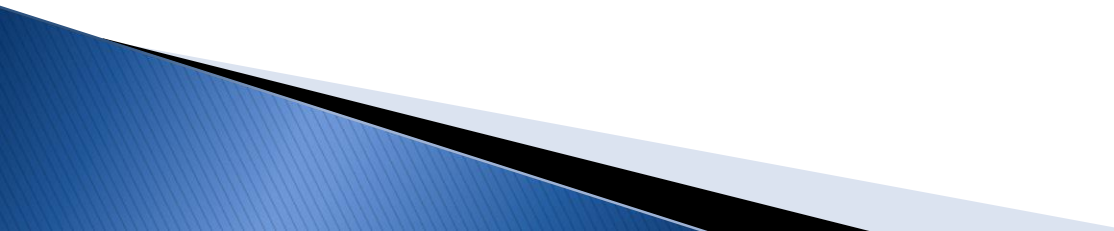
    next++;
    return(next - 1); // Index number
}

void* Stash::fetch(int index) {
    // Check index boundaries:
    assert(0 <= index);
    if(index >= next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(storage[index * size]);
}
```

C++ Stash class

```
int Stash::count() {  
    return next; // Number of elements in CStash  
}  
  
void Stash::inflate(int increase) {  
    assert(increase > 0);  
    int newQuantity = quantity + increase;  
    int newBytes = newQuantity * size;  
    int oldBytes = quantity * size;  
  
    unsigned char* b = new unsigned char[newBytes];  
    for(int i = 0; i < oldBytes; i++)  
        b[i] = storage[i]; // Copy old to new  
  
    delete []storage; // Old storage  
    storage = b; // Point to new memory  
    quantity = newQuantity;  
}
```

C++ class

- ▶ Functions are now inside the structure and are called “member functions”
 - ▶ No need to pass the struct address to each function
 - ▶ No need to name the functions explicitly
 - ▶ Functions have to be declared (usually in the header file) before they can be called
 - ▶ You can access the member variables without referring to the structure
- 

Using the C++ Stash class

- ▶ Variables can be defined at any point in the scope
- ▶ Member functions and variables are selected using (.) operators

```
int main() {  
  
    Stash stash;  
  
    stash.initialize ( sizeof(int) );  
  
    //let's add some elements:  
    for (i = 0; i < 100; i++)  
        stash.add ( &i );  
  
    //4. Now let's print the contents:  
    for(i = 0; i < stash.count(); i++)  
        cout << * ((int*)stash.fetch(i)) << endl;  
  
    stash.cleanup();  
}
```