CSE 165/ENGR 140 Intro to Object Orient Programming

Lecture 7 – Data Abstraction

Announcement

- Quiz #1 on 2/15 (Tuesday) during lecture
- Reading assignment
  - Ch. 5 & 6
  - http://www.cplusplus.com/doc/tutorial/classes/

Recap

- Libraries
  - Class interface (header)
  - Class source code (cpp)
  - Difference between C and C++: C++ is object oriented and has:
    - Encapsulation (Binding data with functions into a class)
    - Data Abstraction (Implementation hiding with an interface)
    - Later we will talk about inheritance and polymorphism

Data Abstraction

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

Wikipedia says:

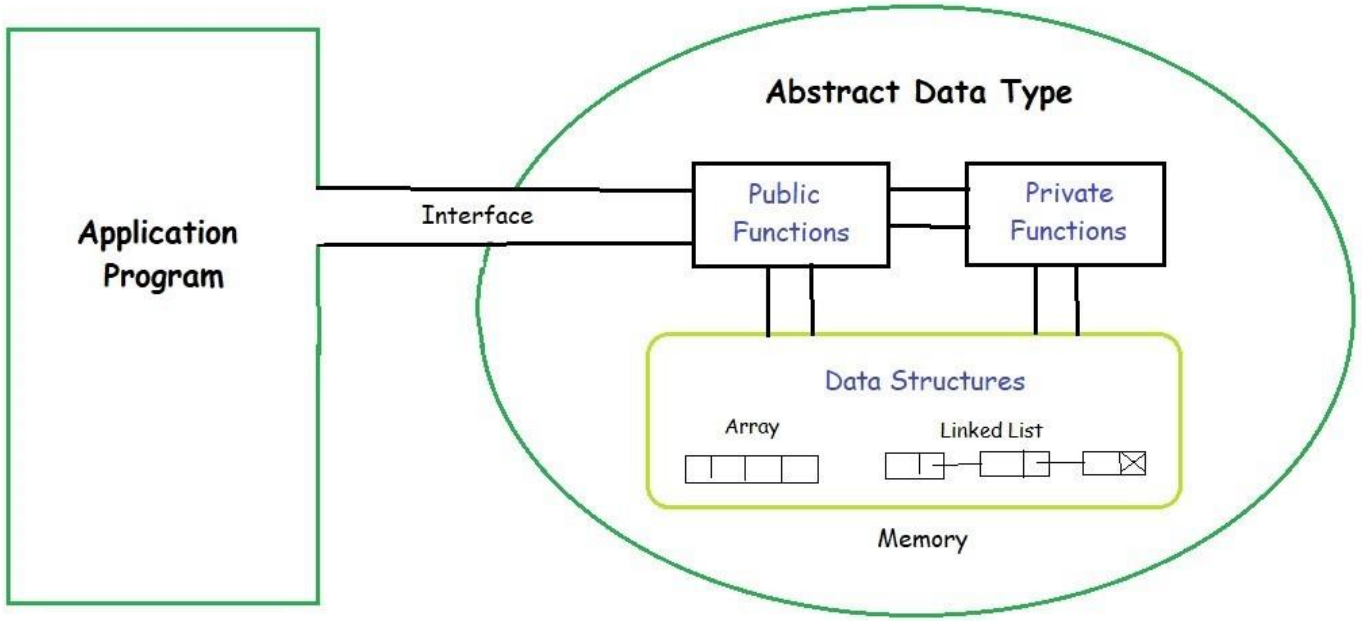
*“In computer science, **abstraction** is the process by which data and programs are defined with **a representation similar in form to its meaning** (semantics), while **hiding away the implementation details**. Abstraction tries to reduce and factor out details so that the programmer can focus on a 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 interface**

Abstract data type (ADT)

- Called “abstract” because it gives an implementation-independent view
- Think of ADT as a black box which hides the inner structure and design of the data type



Abstract data type (ADT)

- We can create new data type by packaging data with functions (using encapsulation)
  - Stash creates a new data type using array
  - Stash has functions to control data (add, fetch, inflate, etc.)
- There are many ADT in the Standard Template Library (STL) of C++
  - Vectors, lists, stacks, queues, etc.

Size of a struct

```
//: C04:Sizeof.cpp
// Sizes of structs
#include <iostream>
using namespace std;

struct A {
   int i[100];
};

struct B {
   void f();
};

void B::f() {}

int main() {
   cout << "sizeof struct A = " << sizeof(A) << " bytes" << endl;
   cout << "sizeof struct B = " << sizeof(B) << " bytes" << endl;
}
```

Output:

sizeof struct A = 400 bytes

sizeof struct B = 1 bytes

Nested structures

```
struct movies {
    string title;
    int year;
};
```

```
struct friends {
    string name;
    string email;
    movies favorite_movie;
} charlie, maria
```

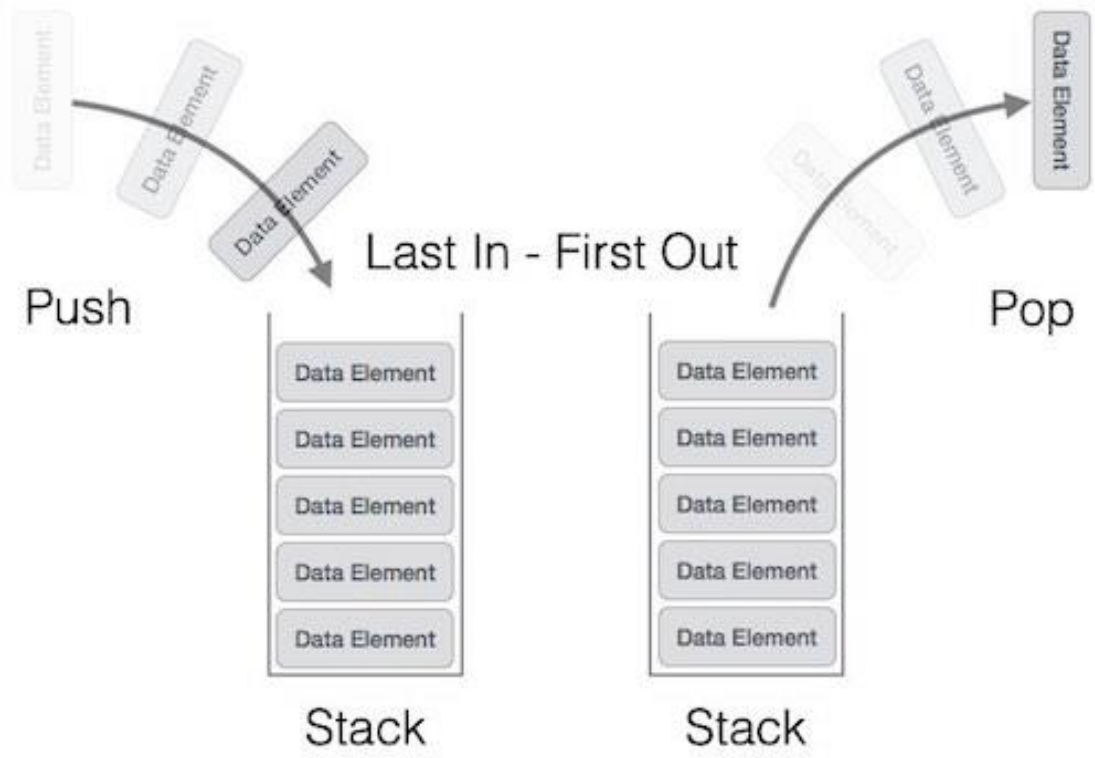
```
friends * p_friends = &charlie;
charlie.name = …
maria.favorite_movie.title = …
charlie.favorite_movie.year = …
p_friends->favorite_movie.title = …
```

Nested structures: Stack example

- What is a ***stack*** of items?
- How are the items in a stack organized?
  - Where do you put a new item (insertion) in a stack?
  - From where do you remove an item (deletion) in a stack?
  - Last-in first-out

Nested structures: Stack example

- We can also organize our data in stacks.



Nested structures: Stack example

- Why do we want to use stacks?
  - History of web browser
  - Un-do function of a text editor
  - Matching “{}” or “()” in a cpp editor

Nested structures: Stack example

```
//: C04:Stack.h
// Nested struct in linked list
#ifndef STACK_H
#define STACK_H

struct Stack {

   struct Link {
      void* data;
      Link* next;
      void initialize(void* dat, Link* nxt);
   }* head;

   void initialize();
   void push(void* dat);
   void* peek();
   void* pop();
   void cleanup();
};

#endif // STACK_H
```

Stack class

```
//: C04:Stack.cpp {O}
#include "Stack.h"
#include "../require.h"
using namespace std;

//Stack::Link has only one method:
void Stack::Link::initialize(void* dat, Link* nxt) {
   data = dat;
   next = nxt;
}
```

Stack class

```
// Stack methods:
void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
   Link* newLink = new Link;
   newLink->initialize(dat, head);
   head = newLink;
}

void* Stack::peek() {
   require(head != 0, "Stack empty");
   return head->data;
}

void* Stack::pop() {
   if(head == 0) return 0;
   void* result = head->data;
   Link* oldHead = head;
   head = head->next;
   delete oldHead;
   return result;
}
```

Stack class

```
// Stack methods:
void Stack::cleanup() {
   // This implementation does not do anything, it just
   // requires the stack to be empty:
    require(head == 0, "Stack not empty");
}

void Stack::cleanup_notused() {
   // We could do something like empty the stack, BUT
   // we do not know the type of the objects stored in the
   // stack, so we cannot free them…
   while ( pop() ); // works, but may create memory leak…
   require(head == 0, "Stack not empty");
}
```

Using stack

```
//: C04:StackTest.cpp
using namespace std;

int main(int argc, char* argv[]) { //Run the program with input arguments
   ifstream in(argv[1]); //input argument as file name
   Stack textlines;
   textlines.initialize();
   string line;

   // Read file and store lines in the Stack:
   while(getline(in, line))
      textlines.push(new string(line));

   // Pop the lines from the Stack and print them:
   string* s;
   while((s = (string*)textlines.pop()) != 0) {
      cout << *s << endl;
      delete s;
   }
   textlines.cleanup();
}
```

Global scope resolution operator

- We use the global scope resolution operator (::, with nothing in front of it) to select a global identifier.

```
//: C04:Scoperes.cpp
// Global scope resolution

   int a;       // 1. Variable a is in the global scope
   void f() {} // 2. Function f is in the global scope

   struct S {
      int a;     // 3. a is a member of S, its global scope is S::a
      void f(); // 4. f is a member of S, its global scope is S::f
   };

   void S::f() {
      ::f();     // 5. Would be recursive otherwise!
      ::a++;     // 6. Select the global a
      a--;       // 7. The a at struct scope
   }
   int main() { S s; f(); s.f(); }
```

Access Control

- 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

```
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
};
```

Access Control

```
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
   }
```

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 ();
 };
```