Review of C++ Basics

Reading:

- King, section 19.4
- Your 181 notes
- Your C++ reference book

Software Design Goals

Suppose we are writing a program that will use several stacks. We are going to implement a stack using

- a struct to hold an array of stack elements plus a counter
- a bunch of functions

Our high-level design should be influenced by several important goals:

- Simple interface.
- Abstraction: The client should not have to know how we implemented the stack in order to use it.
- Information hiding: The client certainly should not be able to touch the data structure directly. We want it to be hidden.
- Encapsulation: The stack code should be self-contained; it shouldn't rely on any other code.
- Plug-out plug-in compatibility: It should be easy to plug in a new and better implementation of stack.
- Easy re-use: It should be easy to use the stack code in another program. Even if we need a slightly different kind of stack.

Using a Class

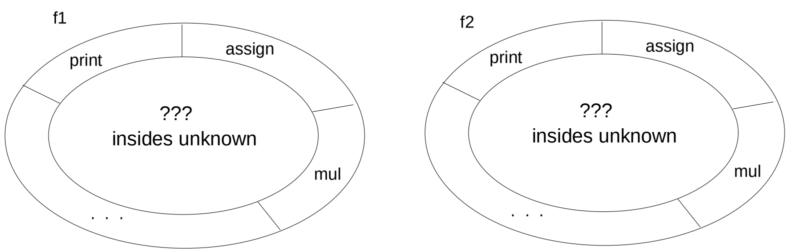
Say we have written a class called Fraction, with operations including `print`. We can now create Fractions much like we create an ordinary int:

```
Fraction f1, f2;
Fraction arrayOfFracs[10];
```

We apply operations to an instance of Fraction using the dot operator:

```
f1.print();          // "f1, Print yourself."
                     // Print has no args.
```

How to Visualize Instances



- You don't have to know what's going on inside in order to use a Fraction. Just how to push the right buttons to get things done.
- Each instance of a Fraction has its own operations.
- Only f1 knows how to print itself (or assign to itself, and so on). Same for f2.

Defining a Class

```
class Fraction {
public:
    Fraction (int num = 0, int denom = 1)
        { assign(num, denom); }
    void print();
    void assign(int num, int denom);
    void mul(Fraction f);
private:
    void reduce();
    int numerator;
    int denominator;
};

void Fraction::assign(int num, int denom)
{
     numerator = num;
     denominator = denom;
     reduce();
}

void Fraction::print ()
{
     printf ("%d/%d", numerator, denominator);
}
```

```
Fraction Fraction::mul (Fraction f)
{
     Fraction result;

     result.numerator = numerator * f.numerator;
     result.denominator = denominator * f.denominator;
     result.reduce();
     return result;
}

void Fraction::reduce()
{
     // details omitted here.
}
```


Using the Class

```
void main()
{
    Fraction f1, f2, f3;
    f1.print();

    f1.assign(3, 8);
    f2.assign(2, 4);
    f1.print();
    f2.print();

    f3 = f1.mul(f2);
    f1.print();
    f2.print();
    f3.print();
}
```

A Class is Like a Struct

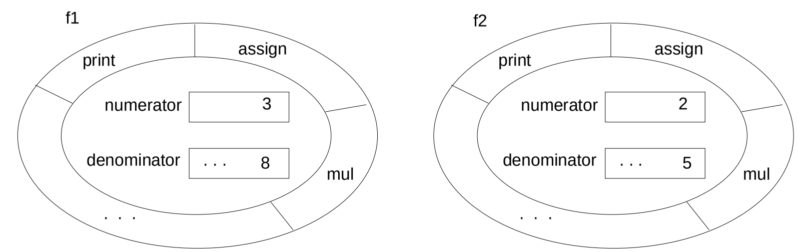
```
struct Fraction {
    int numerator;
    int denominator;
};

void blah ()
{
    Fraction a, b;
    a.numerator = 3;
}
```

```
00  class Fraction {
01  public:
02     ...
03  private:
04     ...
05     // Private data members:
06     int numerator;
07     int denominator;
08  };
09
10  void blah ()
11  {
10      Fraction f1, f2;
11      f1.assign(3, 8);
12  }
```

- In each case, up to line 09, we have defined what a Fraction *is*. But we have no Fractions.
- In each case, At line 10, we create two instances. Memory is allocated to store them.
- Just as a and b each have their own numerator and denominator members, so do f1 and f2.

Visualizing (revisited)



- The private data members (`numerator` and `denominator`) are hidden inside. (1) You don't have to know about them in order to use a Fraction. (2) You *cannot* access them yourself.
- It is the public member functions that have access to the private data members; they do the work for you.
- `f2.print` says "f2, go print yourself". No one else can do it but `f2`.

Two Ways to Create an Instance

```
// --------  M e t h o d  1  ---------------------

struct Tnode {              class Lnode {
    // for a BST                // for a linked list
    ...                         ...
    int num;                    int num;
}                            }

Tnode t;                     Lnode l;
t.num = 12;                  // (OK if public)
                             l.num = 12;
                             l.print();
```

With method (1):

- We immediately get memory for an instance.
- The instance exists only throughout the scope of the variable (in this case, t or l).


```
// --------  M e t h o d  2  ---------------------

struct Tnode {              class Lnode {
    // for a BST                // for a linked list
    ...                         ...
    int num;                    int num;
}                            }

Tnode *pt;                   Lnode * pl;
...                          ...
new pt;                      new pl;
pt -> num = 3;               pl -> num = 3;
                             pl -> print();
```

With method (2):

- We immediately get memory for only a pointer.
- We don't get an instance until we do `new` (like malloc).
- That instance exists, even if the pointer no longer does, throughout the entire program; unless we `delete` it (like free).

Reasons for Using Classes

- Adding a new simple type not in the language.
  E.g., Fractions, complex numbers.
- Creating a variation on a type that *is* in the language.
  E.g., arrays with bounds checking; unlimited-length strings.
- Defining a new kind of composite object — an ADT.
  E.g., stack.
- Separating the "interface" of an ADT from any implementations.
- Using inheritance to define a variation on some other class that we've defined.
  E.g., a searchable stack.

Object-Oriented Programming

In C, we write programs in the "procedural" paradigm.

OOP is more than using classes to support encapsulation. It is a new paradigm.

- We focus on data objects, and the relationships among them
  (vs focusing on tasks).
- We think of data as active — it does things
  (vs having things done *to* it).

Object-oriented programming languages have features that support this vision, and provide some very nice advantages.

Two key features: inheritance and polymorphism.

Describing an ADT's interface

```
// -------------- PriorityQueue.h

# ifndef PRIORITYQUEUE_H
# define PRIORITYQUEUE_H

#include "TodoItem.h"

// A priority queue of TodoItems.
// Limitation: Cannot be used to store objects of any type
// other than TodoItems.

class PriorityQueue {
public:
    // Add td to me, in priority order.
    virtual void enqueue(TodoItem* td) = 0 ;

    // Return my highest priority item, and
    // remove it from me.
    // Precondition: I am not empty.
    virtual TodoItem* dequeue(void) = 0;

    // Return the number of items in me.
    virtual int size() = 0;
};

# endif
```



```
// -------------- LinkedPQ.h

# ifndef LINKEDPQ_H
# define LINKEDPQ_H

#include "PriorityQueue.h"
#include "TodoItem.h"

// Implementation for class PriorityQueue.

class LinkedPQ : public PriorityQueue {
private:
    // Instance variables go here.
public:
    LinkedPQ();

    ~LinkedPQ();

    // Add td to me, in priority order.
    virtual void enqueue(TodoItem *td);

    // Return my highest priority item,
    // and remove it from me.
    // Precondition: I am not empty.
    virtual TodoItem* dequeue();

    // Return the number of items in me.
    virtual int size();
};

# endif
```

```
// ------------- LinkedPQ.cc

#include "LinkedPQ.h"

// Constructor.
LinkedPQ::LinkedPQ() {
    // body omitted
}

// Destructor.
LinkedPQ::~LinkedPQ() {
    // body omitted
}

// Add td to me, in priority order.
void LinkedPQ::enqueue(TodoItem *td) {
   // body omitted
}

// Return my highest priority item,
// and remove it from me.
// Precondition: I am not empty.
TodoItem* LinkedPQ::dequeue() {
   // body omitted
}

// Return the number of items in me.
int LinkedPQ::size() {
   // body omitted
}
```



```
// ------------- Driver.cc

#include <iostream.h>
#include "LinkedPQ.h"
#include "TodoItem.h"

int main() {
   TodoItem td(1, "Fix broken link on 191 web page");
   TodoItem td2(10, "Email Ronnie re Saturday");
   TodoItem td3(3, "Prepare next week's lectures");
   PriorityQueue *pq;
   pq = new LinkedPQ();
   pq->enqueue(&td);
   pq->enqueue(&td2);
   pq->enqueue(&td3);
   int queueSize = pq->size();
   while(queueSize > 0) {
     cout << "Size is: " << pq->size() << endl;
     cout << *pq->dequeue();
     queueSize = pq->size();
   }
}
```

```
// -------------- TodoItem.h

# ifndef TODOITEM_H
# define TODOITEM_H

#include <iostream.h>

class TodoItem {
private:
    static const int DESCR_LENGTH = 10;
    char description[DESCR_LENGTH];
    int priority;

public:
    TodoItem::TodoItem(const int p, const char* d);
    TodoItem::~TodoItem();
    int compareTo(TodoItem other);
    friend ostream &operator<< (ostream& os,
                                const TodoItem& td);
};

# endif
```



```
#include <iostream.h>
#include "TodoItem.h"

// Constructor.
TodoItem::TodoItem(const int p, const char* d) {
    priority = p;
    strcpy(description, d);
}

// Destructor.
TodoItem::~TodoItem() {
    // Nothing to do?
}

int TodoItem::compareTo(const TodoItem other) {

    if (priority < other.priority) {
        // I have higher priority (== a lower priority value)
        return 1;
    } else if (priority > other.priority) {
        return -1;
    } else {
        return 0;
    }
}

ostream& operator<< (ostream& os, const TodoItem& td){
    cout << "Priority: " << td.priority <<
            "; Description: " << td.description;
    cout << endl;
    return os;
}
```