C-Structure

A `struct` is a record that holds together multiple data fields.

Basic syntax:

```
struct structName
{
   dataType1 varName1;
   dataType2 varName2;
   ...
};

structName varInstance; // define an instance of the struct
```

Examples:

```
struct telRecord
{
   string name;  // C++ string object
   string telNo; // http://www.cplusplus.com/reference/string/
};
```

Use the field select operator (`.`) to access a data field of a struct.

Use the arrow operator (`->`) to access a data field of a struct via a pointer.

Examples:

```
telRecord r1, r2;
r1.name = "Peter";
r1.telNo = "98765432";

telRecord *ptr = &r2;      // ptr points to r2
ptr->name = "Amy";         // r2.name = "Amy"
ptr->telNo = "91234567";   // r2.telNo = "91234567"
```

class vs struct

We want to implement a program to support the processing of fraction,

$$fraction = \frac{numerator}{denominator}$$

Requirements of the representation (representation invariants)

- *numerator* and *denominator* are integers
- *denominator* > 0
- *numerator* and *denominator* are relatively prime, $\frac{6}{8}$ is reduced to $\frac{3}{4}$
- only 1 representation of the value zero, $\frac{0}{1}$

Implementation using conventional structural programming

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

unsigned gcd(unsigned m, unsigned n)   // m >= 0 && n > 0
{
   unsigned r;
   while ((r = m % n) > 0)
   {
      m = n;
      n = r;
   }
   return n;
}


//precondition: representation conforms to the requirements
bool isEqual(const fraction& f1, const fraction& f2)
{
   // parameters passed by reference to improve efficiency
   // "const" means that f1 and f2 should not be modified

   return (f1.numerator == f2.numerator) &&
          (f1.denominator == f2.denominator);
}
```

```
fraction addFraction(const fraction& f1, const fraction& f2)
{
   fraction r;
   r.numerator = f1.numerator * f2.denominator +
                 f2.numerator * f1.denominator;

   r.denominator = f1.denominator * f2.denominator;

   int g;
   if (r.numerator >= 0)
      g = gcd(r.numerator, r.denominator);
   else
      g = gcd(-r.numerator, r.denominator);

   r.numerator /= g;
   r.denominator /= g;

   return r;  // return result by value,
              // r ceases to exist after function return
}

// other functions, e.g. subtract, multiple, divide, compare
// to support the manipulation of fraction
```

```
//codes in other functions that use struct fraction

fraction f1, f2, f3, f4; //variable declaration
                         //data fields are not initialized

f1.numerator = 1;   //initialization
f1.denominator = 2;
f2.numerator = 3;
f2.denominator = 4;

f3 = f2; //copy contents of f2 to f3
f4 = addFraction(f1, f2);

fraction *p;       //pointer to a fraction struct
p = new fraction; //dynamic allocation of the memory space
                  //for the fraction struct

p->numerator = 5;  //use -> to access data field via pointer
p->denominator = 6;

f4 = addFraction(f1, *p);
```

Questions:
In the previous example, the function is defined to return the result by value.
What if the `addFraction` function is defined to return the result by reference ?

Case 1:

```
void addFraction(const fraction& f1, const fraction& f2,
                 fraction& r)
// result to be produced is defined as a formal parameter

// What are the necessary changes to the program codes ?
```

Case 2:

```
fraction& addFraction(const fraction& f1, const fraction& f2)
// return a fraction by reference

// What are the necessary changes to the program codes ?
```

Limitation of the conventional structural programming approach

Cannot ensure object instances conform to the required representation invariants

- *numerator* and *denominator* should be relatively prime, and
- *denominator* should be positive

```
fraction f1;
f1.numerator = 6;
f1.denominator = -8;
// value of an object may not conform to the
// representation requirements

fraction f2;
f2.numerator = -3;
f2.denominator = 4;

isEqual(f1, f2) returns false !!
```

Basic syntax of C++ class, constructor and destructor

```
class className
{
protected: // access modifiers: public, protected, private

   DataType dataField;  // member variable

public:

   className();  // default constructor
   ~className(); // destructor

   ReturnType functionName();  // public member function

};
```

```
// Codes in other parts of the program that use the class

if (booleanExpression) // or other control block
{
  className obj;  // data fields of obj are initialized by
                  // the default constructor

  // To invoke a member function on an object instance
  obj.functionName();  // use dot-operator '.'

  // Pointer variable that points to an object instance
  className *p, *q;

  q = &obj; // q is assigned the address of obj;
  p = new className;
  // create an object instance via default constructor
  // using dynamic memory allocation

  p->functionName();  // use -> operator to invoke a function
                      // on the object instance via a pointer
}
// The variable obj goes out-of-scope (not accessible anymore),
// the destructor is invoked on obj
// (statement generated by the compiler automatically).

// Pointer variables p and q also go out-of-scope, but
// the object instance *p remains unchanged !!
```

Modeling of fraction using C++ class

```
#include <ostream>

unsigned gcd(unsigned m, unsigned n)  // m >= 0 && n > 0
{
   unsigned r;
   while ((r = m % n) > 0)
   {
      m = n;
      n = r;
   }
   return n;
}

class fraction
{
   friend ostream& operator<<(ostream& os, fraction& f);
   // overload the operator<< such that you can output
   // a fraction object to an output stream, e.g. cout << f
   // The purpose of this function is similar to the method
   // toString() in Java.

private:
   int numerator;   // member variables
   int denominator; // (instance variables in Java)

public:
   fraction()  // default constructor
   {
      numerator = 0;
      denominator = 1;
   }

   fraction(int n, int d)  // ensure conformant with the
   {                       // representation invariants
      if (d == 0)
      {
         cerr << "ERROR: denominator is zero." << endl;
         exit(0); // terminate the program
      }
      if (n == 0)
      {
         numerator = 0;
         denominator = 1;
      }
      else
```

```
        {
            if (d < 0)
            {
                n = -n;
                d = -d;
            }
            int g;
            if (n < 0)
                g = gcd(-n, d);
            else
                g = gcd(n, d);
            numerator = n / g;
            denominator = d / g;
        }
    }

    //overload (redefine) the operators
    bool operator==(const fraction& other)
    {
        return (this->numerator == other.numerator) &&
               (this->denominator == other.denominator);
        // Pointer "this" points to the implicit object.
    }

    fraction operator+(const fraction& other)
    {
        int n = numerator * other.denominator +
                other.numerator * denominator;

        int d = denominator * other.denominator;
        fraction r(n, d);  //Note that r is a local variable !!
        return r;  //r ceases to exit after function return.
        //It is OK if the function returns by-value.
        //It is an error if the function returns by-reference !!
    }

    void print()
    {
        cout << numerator << "/" << denominator;
    }

    // Other operators and functions in the class.
    // Remark: No need to define a destructor for this class
    // because creation of object instance does not involve
    // dynamic memory allocation.

}; // end of class definition
```

```
// This function must NOT be a member function of the
// class fraction.
// The left operand of operator<< is not a fraction object.
ostream& operator<<(ostream& os, fraction& f)
{
   os << f.numerator << "/" << f.denominator;
   return os;
}

//-------------------------------------------------
// codes in other functions that use class fraction

fraction f1; // f1 is initialized to 0/1 by the
             // default constructor fraction()

fraction f2(6, -8); // f2 is reduced to -3/4 by the
                    // constructor fraction(int, int)

fraction f3 = f1 + f2; // f3.operator=(f1.operator+(f2))

f3.print();  // call the member function print()

cout << f3; // operator<<(cout, f3)

//-------------------------------------------------
fraction *p1 = new fraction;
//create a fraction object with default constructor fraction()

fraction *p2 = new fraction(6, 8);
//use constructor fraction(int, int)

p2->print();  //call member function via a pointer

//-------------------------------------------------
fraction f4 = *p1 + *p2;

fraction f5 = f1;  // copy f1 to f5,
                   // f1 and f5 are two distinct objects

fraction *p3 = p1; // pointers p1 and p3 point to the same
                   // object instance

fraction *p4 = *p1 + *p2; // Error, p4 is undefined !!

fraction *p5 = new fraction;
*p5 = *p1 + *p2;   // OK
```

Remark: object variables in C++ and Java

C++	Java

fraction f1;
//f1 is an object instance
//default constructor is
//invoked automatically
 	
fraction f2(6,8);
	fraction f1, f2;
	//f1 and f2 are object
	//reference
	
	f1 = new fraction();
	//instantiate an object
	//instance
	
	f2 = new fraction(6, 8);

fraction *p1, *p2, *p3;
//pointer to an fraction
//object
 	
p1 = new fraction;
//instantiate an object
//instance
//no bracket for calling
//default constructor
 	
p2 = new fraction(6, 8);
 	
p3 = p1;
//both p3 and p1 point to
//the same object instance
	No explicit pointer variable in Java.

f1 = f2;
//copy contents of f2 to f1
//f1 and f2 are 2 independent
//objects
	f1 = f2;
	//both f1 and f2 refer (point)
	//to the same object instance
	
	f1 = f2.clone();
	//make a copy of f2, and set
	//f1 to point to the newly
	//created copy of f2

Programmer is responsible for the management of dynamically created data objects, i.e. when to release the memory resources back to the system.	Programmer is NOT responsible for the management of dynamically created data objects. The JVM will periodically carry out garbage collection to reclaim the memory resources occupied by unreferenced objects.

Static vs dynamic binding of function calls

- In OO-programming, a derived class inherits the features of the base class.
- In a function call, a class object can be passed by value, by reference, or by pointer.
- C++ allows the user to pass an object of a derived class to a formal parameter of the base class type.

- Static binding (compile-time binding) is used
  - if the object parameter is passed by value, or
  - the member function in the base class is not virtual

- Dynamic binding (run-time binding) is used
  (i) if the object parameter is passed by reference or by pointer, and
  (ii) the member function is a virtual function in the base class.

- A virtual function in the base class will have an implementation.
- A virtual function without an implementation is called a pure virtual function.
- Pure virtual function in C++ corresponds to abstract method in Java.
- Java always uses dynamic binding (run-time binding) in method invocation.

- Execution speed using static binding is faster. C-language only supports static binding.
- Advantages of dynamic binding
  - Allows subtype polymorphism and class-specific methods
  - Allows new subclasses to be added without modifying clients
- Disadvantages of dynamic binding
  - Makes logic of program harder to follow
  - Adds run-time overhead: time to do method lookup, and space for run-time class information.

Examples:

```
class baseClass
{
private:
   int x;

public:
   baseClass(int u = 0)
   {
      x = u;
   }

   virtual void print()
   {
      cout << "In baseClass x = " << x << endl;
   }
};
```

```
//derivedClass extends (or inherits) baseClass
class derivedClass : public baseClass
{
private:
   int a;

public:
   derivedClass(int u = 0, int v = 0) : baseClass(u)
   {  // invoke the baseClass constructor to initialize x

      a = v;
   }

   void print()  // override the print() function
   {
      cout << "derivedClass::print()" << endl;
      baseClass::print();
      cout << "In derivedClass a = " << a << endl;
   }

};
```

```
//Case 1:

void callPrint(baseClass& b)  // b is passed by reference
{
   b.print();  // dynamic binding is used
}

int main()
{
   baseClass baseObj(5);             // x = 5
   derivedClass derivedObj(3, 8);  // x = 3, a = 8

   callPrint(baseObj);
   callPrint(derivedObj); // print() function of derived class
                          // is executed
}
```

```
Output:
In baseClass x = 5
derivedClass::print()
In baseClass x = 3
In derivedClass a = 8
```

```
//Case 2:

void callPrint(baseClass *p)  //p is passed by pointer
{
   p->print();  //dynamic binding is used
}

int main()
{
   baseClass baseObj(5);
   derivedClass derivedObj(3, 8);

   callPrint(&baseObj);
   callPrint(&derivedObj); //print() function of derived class
                           //is executed
}
```

```
Output:
In baseClass x = 5
derivedClass::print()
In baseClass x = 3
In derivedClass a = 8
```

//Case 3:

```
void callPrint(baseClass b)  //b is passed by value
{
   b.print();  //static binding is used
}

int main()
{
   baseClass baseObj(5);
   derivedClass derivedObj(3, 8);

   callPrint(baseObj);
   callPrint(derivedObj); //print() function of baseClass is
                          //executed
}
```

```
Output:
In baseClass x = 5
In baseClass x = 3
```

Example: Array-based List (similar to ArrayList in Java)

List : A collection of elements of the same type.

Operations on a list
1. Create a list (empty list)
2. Determine if the list is empty.
3. Determine if the list is full.
4. Find the size (length) of the list.
5. Destroy, or clear, the list.
6. Determine whether an item is contained in the list.
7. Insert an item (at the default location or a given location).
8. Remove an item (from the default location, or a given location, or given value).
9. Retrieve an item from the list (at the default location, or the specified location).
10. Search the list for a given item.

Implementation consideration 1:

How to store the list in the computer's memory?

- In this example, we store the list items using an array.
- Storage space of the array is created dynamically on-demand.

Implementation consideration 2:

We want the program codes to be reusable for different data types,
e.g. `int`, `double`, `fraction`, and etc.

- Develop generic codes using **`template`**
- In Java, this is called generic class (or generic programming)

```
// filename: arrayListType.h

#ifndef ARRAY_LIST_TYPE_H
#define ARRAY_LIST_TYPE_H

template<class elemType>  //elemType is a type parameter
class arrayListType
{
protected:
   elemType *list; //array to hold the elements
   int length;     //no. of elements in the list
   int maxSize;    //physical size of array

public:
   arrayListType(int size = 100);
   //constructor, default size = 100

   arrayListType(const arrayListType<elemType>& other);
   //copy constructor

   ~arrayListType();
   //destructor, deallocate the memory occupied by the array
   //A user defined destructor is necessary if the object
   //instance contains dynamically allocated memory.

   //getter (accessor) functions
   bool isEmpty() const;
   bool isFull() const;
   int listSize() const;
   int maxListSize() const;
   void print() const;
   void retrieve(int loc, elemType& item);
   virtual int search(const elemType& item);

   //setter (mutator) functions
   arrayListType<elmType>& operator=
                        (const arrayListType<elmType>& other);
   void clearList();
   virtual void insert(const elemType& item);
   virtual void remove(const elemType& item);
   virtual void removeAt(int loc);

//search, insert, remove, removeAt are virtual functions.
//Their implementations can be overridden in derived classes.

};  //end of class definition
```

```
//Implementations of the functions in the template class
//should be put in the .h file of the template class

template<class elemType>
arrayListType<elemType>::arrayListType(int size)
{
   if (size < 0)
   {
      cerr << "List size must be positive" << endl;
      maxSize = 100;
   }
   else
      maxSize = size;

   list = new elemType[maxSize]; //dynamic memory allocation

   assert (list != nullptr);
   //For development (debugging) phase.
   //Assertion test is usually turned off during
   //production phase.
}

//overload the copy constructor
template<class elemType>
arrayListType<elemType>::arrayListType
                        (const arrayListType<elemType>& other)
{
   //The copy constructor is called when an object is passed
   //as a value parameter to a function.

   maxSize = other.maxSize;
   length = other.length;

   //make a copy of the array
   list = new elemType[maxSize];
   assert(list != nullptr);

   for (int i = 0; i < length; i++)
      list[i] = other.list[i];
}


template<class elemType>
arrayListType<elemType>::~arrayListType()
{
   delete[] list; //free the memory occupied by the array
}
```

```
//overload (redefine) the assignment operator
template<class elemType>
arrayListType<elemType>& arrayListType<elmType>::operator=
                      (const arrayListType<elemType>& other)
{
   if (this != &other) //"this" is the pointer that points to
   {                    //the implicit object
      length = other.length;

      if (maxSize != other.maxSize)
      {
         maxSize = other.maxSize;
         delete[] list;
         list = new elemType[maxSize];
         assert(list != nullptr);
      }

      for (int i = 0; i < length; i++) //copy list elements
         list[i] = other.list[i];
   }
   return *this;  //return the implicit object (by reference)
}

template<class elemType>
bool arrayListType<elemType>::isEmpty() const
{
   return length == 0;
}

template<class elemType>
bool arrayListType<elemType>::isFull() const
{
   return length >= maxSize;
}

template<class elemType>
int arrayListType<elemType>::listSize() const
{
   return length;
}

template<class elemType>
int arrayListType<elemType>::maxListSize() const
{
   return maxSize;
}
```

```
template<class elemType>
void arrayListType<elemType>::print() const
{
   for (int i = 0; i < length; i++)
      cout << list[i] << " ";

   cout << endl;
}

template<class elemType>
int arrayListType<elemType>::search(const elemType& item)
{
   // item is passed by reference to improve efficiency
   for (int i = 0; i < length; i++)
      if (list[i] == item)
         return i;

   return -1; //item not found
}

template<class elemType>
void arrayListType<elemType>::
                          retrieve(int loc, elemType& item)
{
   if (loc < 0 || loc >= length)
      cerr << "Error: Index out of bound" << endl;
   else
      item = list[loc];
}

template<class elemType>
void arrayListType<elemType>::clearList()
{
   length = 0;
}

template<class elemType>
void arrayListType<elemType>::insert(const elemType& item)
{
   if (!isFull())
      list[length++] = item;
}
```

```
template<class elemType>
void arrayListType<elemType>::remove(const elemType& item)
{
   int i = 0;
   bool done = false;

   while (i < length && !done)
   {
      if (list[i] == item)
      {
         list[i] = list[--length];
         done = true;
      }
      else
         i++;
   }
}

template<class elemType>
void arrayListType<elemType>::removeAt(int loc)
{
   if (loc < 0 || loc >= length)
      cerr << "Error: Index out of bound" << endl;
   else
      list[loc] = list[--length];
}


#endif //end of the .h file
```

```
//Example program that makes use of arrayListType

int main()
{
   arrayListType<int> intList(20);

   arrayListType<fraction> fractionList(50);

   for (int i = 1; i < 20; i++)
      intList.insert(i);

   intList.print();

   for (int t = 1; t < 50; t++)
   {
      fraction f(t, t+1);
      fractionList.insert(f);
   }

   fractionList.print();

   ...
}
```

Remark about the compiler regarding the `operator=`

```
arrayListType<int> list_1(20);

for (int i = 0; i < 20; i++)
   list_1.insert(i);

arrayListType<int> list_2;
//list_2 created by default constructor

list_2 = list_1;  //use the overloaded operator=


arrayListType<int> list_3 = list_1; //initialization
//list_3 is created by direct copying of the member variables
//of list_1
//The overloaded operator= is NOT used !!
```

Elements of the array-based list in the above example are unordered.
We can implement an ordered list using inheritance.

```
// filename: orderedArrayListType.h
#ifndef ORDERED_ARRAY_LIST_TYPE_H
#define ORDERED_ARRAY_LIST_TYPE_H

#include "arrayListType.h"

template <class elemType>
class ordered_arrayListType : public arrayListType<elemType>
{
   //no additional variables required in this example

public:
   ordered_arrayListType() : arrayListType()
   { }

   ordered_arrayListType(int n) : arrayListType(n)
   { }

   //override the virtual functions in base class
   int search(const elemType& item);
   void insert(const elemType& item);
   void remove(const elemType& item);
   void removeAt(int loc);
};

template<class elemType>
int ordered_arrayListType<elemType>::
                                    search(const elemType& item)
{  //use binary search
   int low = 0;
   int high = length-1;
   while (low <= high)
   {
      int mid = (low + high)/2;
      if (list[mid] == item)
         return mid;
      if (list[mid] < item)
         low = mid + 1;
      else
         high = mid - 1;
   }
   return -1; //item not found
}
```

```
//mutator functions should maintain the ordering of elements
//in the list

template<class elemType>
void ordered_arrayListType<elemType>::
                                        insert(const elemType& item)
{
   if (isFull())
      return;

   int i;  // i must be declared outside the for-loop !!
   for (i = length-1; i >= 0 && list[i] > item; i--)
      list[i+1] = list[i];

   list[i+1] = item;
   length++;
}


template<class elemType>
void ordered_arrayListType<elemType>::
                                        remove(const elemType& item)
{
   int k = search(item);
   if (k >= 0)
   {
      for (int i = k; i < length-1; i++)
         list[i] = list[i+1];
      length--;
   }
}


template<class elemType>
void ordered_arrayListType<elemType>::removeAt(int loc)
{
   if (loc >= 0 && loc < length)
   {
      for (int i = loc; i < length-1; i++)
         list[i] = list[i+1];
      length--;
   }
}

#endif
```

Remarks on array-based list

1. The size of the physical array is fixed during execution. The physical size of the array and the logical length of the list are two different attributes.
2. New items can be added to the array only if there is room.
3. Expansion of an array is possible – create a new array and copy the contents of the old array to the new array (this option is used by Java in the `class ArrayList<E>`). But this operation is time consuming.
4. If the array is ordered, then searching can be done more efficiently using binary search.
5. Insertion and deletion operation on an ordered array is time consuming, i.e. requires shifting of the array elements to maintain the ordering.
6. If `elemType` is a user defined class (e.g. `fraction`), then the programmer needs to overload all the arithmetic and relational operators in the user defined class !!