C-Structure

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

Basic syntax:

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

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

Examples:

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

Remark:
Unlike `cstring` (char array in C), `class string` in C++ has built-in support to handle dynamic memory allocation and deallocation.

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"
```

Structural programming in C vs OOP in C++

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 normalized 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;
};

int gcd(int m, int n)
{
   if (m < 0) m *= -1;
   if (n < 0) n *= -1;

   int r;
   while ((r = m % n) > 0)
   {
      m = n;
      n = r;
   }
   return n;
}


//precondition: representation conforms to the requirements
bool equal(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 = 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); // 4/3 = 1/2 + 5/6
```

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;

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

Template function (generic programming) in C++ can be used to process data of different "compatible" types.

```
template<class Type>
int search(Type* a, int n, Type key) // search an unordered
{                                    // array
   for (int i = 0; i < n; i++)
      if (a[i] == key)
         return i;
   return -1;

}
```

The above template function can be used to search an unordered array of different data types, e.g. `float`, `double`, `int`, `long long`, `string`, or user defined class.

The prerequisite is that all operators used in the template function are properly supported (i.e. defined in the programming language or implemented in the class concerned).

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 (required if the object
                 // contains dynamically allocated memory
                 // resource)

   ReturnType functionName();  // public member function

};
```

Remark:

Usage of **struct** in C++ has been generalized to represent a **class** with all member variables and functions being public.

```
struct className
{
    DataType dataField;          // public member variable

    className();                 // public constructor
    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),
// 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>
class fraction
{
   friend ostream& operator<<(ostream& os, const 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, no explicit parameter
   {
      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 = 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.

   }

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

   // Other operators and functions in the class (not shown).
   // 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, const 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 normalized 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 memory space used by dynamically created data objects. The JVM will periodically carry out garbage collection to reclaim the memory resources occupied by unreferenced objects.