

Miscellany on Functions & Pointers

Umesh Bellur

Recap

- Defining a struct
 - Members
 - Data
 - Functions
- Declaring Struct instances
- Accessing struct members
- Initializing and assignments to struct instances.
- Passing structs to functions
 - By value
 - By reference/pointer

Today's topics

- Function pointers
- Default parameter values in functions
- Function overloading
- Back to Structures and member functions
 - Constructors
 - Destructors
 - Operators
 - Unary and Binary

Function Pointers

Question

- How do you sort an array of anything?
 - You do not know its an array of ints or an array of strings or an array of students or
- What is the operation fundamental to sorting?

qsort

- A unix utility function that can be used to sort any data set stored in an array (really cool!!)

NAME

qsort - sorts an array

SYNOPSIS

```
#include <stdlib.h>
```

```
void qsort(void *base, size_t nmemb, size_t size,  
           int(*compar)(const void *, const void *));
```

DESCRIPTION

The `qsort()` function sorts an array with `nmemb` elements of size `size`. The `base` argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

RETURN VALUE

The `qsort()` function returns no value.

How do you

- Write code that prints a double value X and some function of X that is specified dynamically.
 - Eg: the function may be a square of X
 - Eg: the function may be a square root of X
 - Etc.

What are function Pointers?

- C/C++ does not require that pointers only point to data, it is possible to have pointers to functions
- Functions occupy memory locations therefore every function has an address just like each variable

Code and Data

```
int foo( ){  
    .....  
}
```

```
int main (int argc, char* argv[])  
{ foo();  
    .....  
}
```

↓
Assembly code

```
main:  
.LFB6:  
    pushq %rbp  
.LCFI2:  
    movq %rsp, %rbp  
.LCFI3:  
    subq $32, %rsp  
.LCFI4:  
    movl %edi, -20(%rbp)  
    movq %rsi, -32(%rbp)  
    movl $10, -4(%rbp)  
    movl -4(%rbp), %edi  
    call foo  
    leave  
    ret
```



Why do we need function Pointers?

- Useful when ***alternative functions maybe used to perform similar tasks on data*** (eg sorting integers vs sorting strings)
- One common use is in passing a function as a parameter in a function call.
 - Can pass the data and the function to be used to some control function
- Greater flexibility and better code reuse

Define a Function Pointer

- A function pointer is nothing but a variable, it must be defined as usual.

```
int (*myFunPointer) (int, char, int);
```

- ***myFunPointer*** is a pointer to a function that accepts 3 arguments of types int, char and int respectively and returns a result of type int.
 - The extra parentheses around (*funcPointer) is needed because there are precedence relationships in declaration just as there are in expressions

Assign an address to a Function Pointer

- It is optional to use the address operator & in front of the function's name
- When you mention the name of a function but are not calling it, there's nothing else you could possibly be trying to do except for generating a pointer to it
- Similar to the fact that a pointer to the first element of an array is generated automatically when an array appears in an expression

Assign an address to a Function Pointer

```
//assign an address to the function pointer
int (*myFunPointer) (int, char, int);

int firstExample ( int a, char b, int c){
    return a+b+c;
}
myFunPointer= firstExample; //assignment
myFunPointer=&firstExample; //alternative
```

Comparing Function Pointers

- Can use the (==) operator

```
//comparing function pointers  
if(myFunPointer == &firstExample)  
    cout << "pointer points to firstExample";
```

Calling a function using a Function Pointer

There are two alternatives

- 1) Use the name of the function pointer
- 2) Can explicitly dereference it

```
int (*funcPointer) (int, char, int);  
// calling a function using function  
// pointer  
int answer= funcPointer (7, 'A' , 2 );  
int answer= (*funcPointer) (7, 'A' , 2 );
```

Example: Trigonometric Functions

```
// prints tables showing the values of cos,sin

void tabulate(double (*f)(double), double first, double last, double incr);

int main() {
    double final, increment, initial;

    cout << "Enter initial value:" << endl;
    cin >> initial;

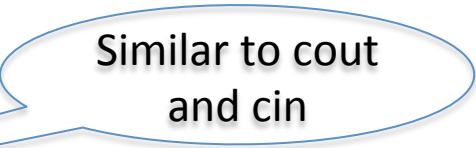
    printf ("Enter final value: "); // similar to cout in C
    scanf ("%lf", &final);         // similar to cin in C

    cin >> increment;

    tabulate(cos, initial,final,increment);

    tabulate(sin, initial,final,increment);

    return 0;
}
```



Similar to cout
and cin

Trigonometric Functions

```
// when passed a pointer f prints a table showing the
// value of f
void tabulate(double (*f) (double),
              double first, double last, double incr){

    double x;
    int i, num_intervals;

    num_intervals = ceil ( (last -first) /incr );

    for (i=0; i<=num_intervals; i++){
        x= first +i * incr;
        cout << x << (*f) (x) << endl;
    }
}
```

Enter initial value: 0

Enter final value: .5

Enter increment: .1

0.00000	1.00000
0.10000	0.99500
0.20000	0.98007
0.30000	0.95534
0.40000	0.92106
0.50000	0.87758



For $\cos(x)$

0.00000	0.00000
0.10000	0.09983
0.20000	0.19867
0.30000	0.29552
0.40000	0.38942
0.50000	0.47943



For $\sin(x)$

Sorting

Consists of three parts

- 1) a comparison that determines the ordering of any pair of objects
- 2) an exchange that reverses their order
- 3) A sorting algorithm that makes comparisons and exchange until the objects are in order.

The sorting algorithm is independent of the comparison and exchange operator

- **qsort** will sort an array of elements. This is a wildcard function that uses a pointer to another function that performs the required comparisons.
- Library: `stdlib.h` Prototype:

```
void qsort (void *base, size_t num, size_t size,  
            int (*comp_func) (const void *, const void *))
```

- **Some explanation.**
 - `void * base` is a pointer to the array to be sorted. This can be a pointer to any data type
 - `size_t num` The number of elements.
 - `size_t size` The element size.
 - `int (*comp_func)(const void *, const void *)` This is a pointer to a function.

Sorting

- qsort thus maintains it's **data type independence** by giving the comparison responsibility to the user.
- The compare function must return integer values according to the comparison result:
 - less than zero : if first value is less than the second value
 - zero : if first value is equal to the second value
 - greater than zero : if first value is greater than the second value
- Complicated data structures can be sorted in this manner.
- The generic pointer type void * is used for the pointer arguments, any pointer can be cast to void * and back again without loss of information.

Qsort for Asc and Des order of Strings

```
#include <stdio.h>
#define NSTRS 10 /* number of strings */
#define STRLEN 16 /* length of each string */
char strs [NSTRS] [STRLEN]; /* array of strings */
int compare1(char*, char*);
int compare2(char*, char*);
main() {
    int i;

    /* Prompt the user for NSTRS strings. */
    for (i = 0; i < NSTRS; i++) {
        cout << "Enter string #" << i << ":";
        cin.getline(strs[i]);
    }

    /* * Sort the strings into ascending order.
    There are NSTRS array elements, each one is STRLEN characters long.*/

    qsort(strs, NSTRS, STRLEN, compare1);

    /* * Now sort the strings in descending order. */
    qsort(strs, NSTRS, STRLEN, compare2);
```

One for
ascending and
one for
descending

Pass function ptr to
one that returns
compare of a>b

qsort

```
/* * compare1--compare a and b, and return less than  
    greater than, or equal to zero. Since  
    we are comparing character strings, we  
    can just use strcmp to do the work for us. */
```

```
int compare1(char* a, char* b) {  
    return(strcmp(a, b));  
}
```

```
/* * compare2--this compares a and b, but is used for  
    * sorting in the opposite order. Thus it  
    returns the opposite of strcmp. We can  
    * simulate this by simply reversing the  
    * arguments when we call strcmp. */
```

```
int compare2(char* a, char* b) {  
    return(strcmp(b, a));  
}
```

Arrays of Function Pointers

- C treats pointers to functions just like pointers to data therefore we can have arrays of pointers to functions
- This offers the possibility to select a function using an index
 - Eg: suppose that we're writing a program that displays a menu of commands for the user to choose from. We can write functions that implement these commands, then store pointers to the functions in an array:

```
void (*file_cmd[]) (void) =  
{ new_cmd,  
  open_cmd,  
  close_cmd,  
  save_cmd ,  
  save_as_cmd,  
  print_cmd,  
  exit_cmd  
};
```

If the user selects a command between 0 and 6, then we can subscript the file_cmd array to find out which function to call

```
file_cmd[n] ();
```


Another Example

```
void add(int a, int b) {  
    cout << "addition: " << a+b << endl;  
}  
void subtract(int a, int b){  
    cout << "subtraction: " << a-b << endl;  
}  
void multiply(int a, int b){  
    cout << "mult: " << a*b << endl;  
}
```

The functions

```
int main(){  
    // fun_ptr_arr is an array of function pointers  
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};  
  
    unsigned int ch, a, b;  
    cout << "Enter Choice: 0 for add, 1 for subtract and 2 "  
        << "for multiply" << endl;  
    cin >> ch;  
  
    if (ch > 2) return 0;  
    (*fun_ptr_arr[ch])(a, b);  
    return 0;  
}
```

An Array of fn
pointers

Calling the
appropriate fn

Function Pointers and Ordinary Pointers

- C++ makes a clear distinction between the two types of pointers
- A data pointer only hold the address of the first byte of a variable
- While function pointers share many similarities with ordinary data pointers, they differ in several ways.
 - First, you can't declare a generic pointer to function similar to `void*`.
 - In addition, trying to store a function pointer in a `void*` isn't guaranteed to work (certain compilers tolerate this, others don't).
 - Finally, you can't dereference a function pointer -- there's no such thing as passing a function by value.

Default parameter values

Default Arguments

- Some functions can take several arguments
 - Can increase function flexibility
 - Can reduce proliferation of near-identical functions
- But, callers must supply all of these arguments
 - Even for ones that aren't "important"
- We can provide defaults for some arguments
 - Caller doesn't have to fill these in

Required vs. Default Arguments

- Function with required argument

```
// call as foo(2); (prints 2)
void foo(int a);
void foo(int a) {cout << a << endl;}
```

- Function with default argument
 - Notice only the *declaration* gives the default value

```
// can call as foo(2); (prints 2)
// or can call as foo(); (prints 3)
void foo(int a = 3);
void foo(int a) {cout << a << endl;}
```

Defaults with Multiple Arguments

- Function with one of two arguments defaulted

```
void foo(int a, int b = 3);  
void foo(int a, int b)  
    {cout << a << " " << b << endl;}  
// can call as foo(2); (prints 2 3)  
// or can call as foo(2, 4); (prints 2 4)
```

- Same function, with both arguments defaulted

```
void foo(int a = 1, int b = 3);  
void foo(int a, int b)  
    {cout << a << " " << b << endl;}  
  
// can call as foo(); (prints 1 3)  
// or can call as foo(2); (prints 2 3)  
// or can call as foo(2, 4); (prints 2 4)
```

Default Argument Limitations

- Watch out for ambiguous signatures
 - `foo()`; and `foo(int a = 2)`; for example
- Can only default the rightmost arguments
 - Can't declare `void foo(int a = 1, int b)`;

Function Overloading

Function Overloading

A ***function*** name having several definitions that are differentiable by the number or types of their arguments. The number and types of arguments along with their order makes up the ***signature*** of the function

.

For example;

```
float divide (int a, int b);  
float divide (float x, float y);
```

If two functions are having same number and types of arguments in the same order, they are said to have the same *signature*. Even if they are using distinct variable names, it doesn't matter. For instance, following two functions have same signature.

```
void squar (int a, float b); //function 1  
void squar (int x, float y); //same function as  
                             that of function 1
```

To overload a function name, all you need to do is, ***declare and define all the functions with the same name but different signatures, separately***. For instance, following code fragment overloads a function name **prnsqr()**.

```
void prnsqr (int i);           //overloaded for integer #1
void prnsqr (char c);         //overloaded for character #2
void prnsqr (float f);        //overloaded for floats #3
void prnsqr (double d);       //overloaded for double floats #4
```

After declaring overloading functions, you must define them separately, as it is shown below for above given declarations.

```
void prnsqr (int i){  
    cout<<"Integer"<<i<<"'s square is"<<i*i<<"\n";  
}  
void prnsqr (char c);{  
    cout<<c<<"is a character"<<"Thus No Square for it"<<"\n";  
}  
void prnsqr (float f){  
    cout<<"float"<<f<<"'s square is" << f*f<< endl;  
}  
void prnsqr (double d){  
    cout <<"Double float" << d <<"'s square is" << d*d << endl;  
}
```

Thus, we see that is not too difficult in declaring overloaded functions; they are declared as other functions are. Just one thing is to be kept in mind that the arguments are sufficiently different to allow the functions to be differentiated in use.

The argument types are said to be part of function's extended name. For instance, the name of above specified functions might be **prnsqr()** but their extended names are different. That is they have **prnsqr(int)**, **prnsqr(char)**, **prnsqr(float)**, and **prnsqr(double)** extended names respectively.

When a function name is declared more than once in a program, the compiler will interpret the second (and subsequent) declaration(s) as follows:

- 1) If the signatures of subsequent functions match the previous function's, then the second is treated as a re-declaration of the first.
- 2) If the signatures of two functions match exactly but the return type differ, the second declaration is treated as an erroneous re-declaration of the first and is flagged at compile time as an error.

For example,

```
float square (float f);  
double square (float x);    //error
```

Functions with the same signature and same name but different return types are not allowed in C++. You *can* have different return types, but only if the signatures are also different:

```
float square (float f);    //different signatures, hence  
double square (double d); //allowed
```

If the signature of the two functions differ in either the number or type of their arguments, the two functions are considered to be *overloaded*.



Use function overloading only when a function is required to work for alternative argument types and there is a definite way of optimizing the function for the argument type.

Restrictions on Overloaded Functions

Several restrictions governs an overloaded functions:

- Any two functions in a file must have different signatures.
- Overloading functions with same types, based on return type, is an error.
- Member functions cannot be overloaded solely on the basis of one being static and the other nonstatic.

Static functions are those that are confined to the file they are defined in OR in the case of member functions they can be called without an object or instance of the struct existing.

- Typedef declaration do not define new types; they introduces synonyms for existing types. They **do not affect** the overloading mechanism. Consider the following code:

```
typedef char* PSTR;  
void Print (char * szToPrint);  
void Print (PSTR szToPrint);
```

CALLING OVERLOADED FUNCTIONS

Overloaded functions are called just like other functions. *The number and type of arguments determine which function should be invoked.*

For instance consider the following code fragment:

```
prnsqr ( 'z' );  
prnsqr (13) ;  
prnsqr (134.520000012) ;  
prnsqr (12.5F) ;
```

Steps Involved in Finding the Best Match

A call to an overloaded function is resolved to a particular instance of the function through a process known as *argument matching*, which can be termed as a *process of disambiguation*. Argument matching involves comparing the actual arguments of the call with the formal arguments of each declared instance of the function. There are three possible cases, a function call may result in:

- a) A match.** A match is found for the function call.
- b) No match.** No match is found for the function call.
- c) Ambiguous Match.** More than one defined instance for the function call.

1. Search for an Exact Match

If the type of the actual argument exactly matches the type of one defined instance, the compiler invokes that particular instance. For example,

```
void afunc(int);      //overloaded functions
void afunc(double);
afunc(0);             //exactly match. Matches afunc(int)
```

0 (zero) is of type **int** , thus the call exactly matches **afunc(int)**.

2. A match through **promotion**

If no exact match is found, an attempt is made to achieve a match through promotion of the actual argument.

Recall that the conversion of integer types (**char**, **short**, **enumerator**, **int**) into **int** (if all values of the type can be represented by **int**) or into **unsigned int** (if all values can't be represented by **int**) is called *integral promotion*.

For example, consider the following code fragment:

```
void afunc (int);  
void afunc (float);  
afunc ( 'c' );    //match through the promotion;  
                  matches afunc (int)
```

3. A match through application of standard C++ conversion rules

If no exact match or match through a promotion is found, an attempt is made to achieve a match through a standard conversion of the actual argument. Consider the following example,

```
void afunc (char);  
void afunc (double);  
afunc (471);           //match through standard  
                        conversion matches afunc  
                        (double)
```


The **int** argument 471 can be converted to a **double** value 471 using C++ standard conversion rules and thus the function call matches (through standard conversion) **func(double)**.

But if the actual argument may be converted to multiple formal argument types, the compiler will generate an error message as it will be an ambiguous match. For example,

```
void afunc (long);  
void afunc (double);  
afunc(15);           //Error !! Ambiguous match
```

Here the **int** argument 15 can be converted either **long** or **double**, thereby creating an ambiguous situation as to which **afunc()** should be used.

Any function, whether it is a class member or just an ordinary function can be overloaded in C++, provided it is required to work for distinct argument types, numbers and combinations.

Default Arguments Versus Overloading

Using default argument gives the appearance of overloading, because the function may be called with an optional number of arguments.

For instance, consider the following function prototype:

```
float amount (float principal, int time=2,  
              float rate=0.08);
```

Now this function may be called by providing just one or two or all three argument values. A function call like as follows:

```
cout << amount (3000);
```

will invoke the function **amount()** with argument values 3000, 2, and 0.08 respectively. Similarly a function call like

```
cout << amount (3000, 4);
```

Will invoke **amount()** with argument values 3000, 4, and 0.08 respectively. That is if argument values are provided with the function call, then the function is invoked with the given values. But if any value is missing and there has been default values specified for it, then the function is invoked using the default value.

However if you skip the middle argument **time** but C++ makes no attempt at this type of interpretation.

```
amount(3000, 0.13);
```

What is called?

Amount with 3000, 0 and 0.08

That is, with default arguments C++ expects that only the arguments on the right side can be defaulted. If you want to default a middle argument, then all the arguments on its right must also be defaulted.

Back to Member functions

Motivational Example: The Queue Struct in Taxi Dispatch

```
const int N=100;
struct queue{
    int elements[N],
        nwaiting,front;
    bool insert(int v){
        ...
    }
    bool remove(int &v){
        ...
    }
};
```

- Once the queue is created, we expect it to be used only through the member functions, insert and remove
- We do not expect elements, nWaiting, front to be directly accessed

Main Program Using Queue

```
int main(){
    Queue q;
    q.front = q.nWaiting = 0;
    while(true){
        char c; cin >> c;
        if(c == 'd'){
            int driver; cin >> driver;
            if(!q.insert(driver))
                cout << "Q is full\n";
        }
        else if(c == 'c'){
            int driver;
            if(!q.remove(driver))
                cout << "No taxi.\n";
            else cout << "Assigning <<
                driver<< endl;
        }
    }
}
```

- Main program does use q through operations insert and remove
- However, at the beginning, q.front and q.nWaiting is directly manipulated
- This is against the philosophy of software packaging
- When we create a queue, we will always set q.nWaiting and q.front to 0
- C++ provides a way by which the initialization can be made to happen automatically, and also such that programs using Queue do not need to access the data members directly
- Just defining Queue q; would by itself set q.nWaiting and q.front to 0!

— Next

Constructor Example

- In C++, the programmer may define a special member function called a **constructor** which will always be called when an instance of the struct is created
- A constructor has the same name as the struct, and no return type
- The code inside the constructor can perform initializations of members
- When q is created in the main program, the constructor is called automatically

```
struct Queue{  
    int elements[N], front,  
        nWaiting;  
    Queue(){ // constructor  
        nWaiting = 0;  
        front = 0;  
    }  
    // other member functions  
};  
  
int main(){  
    Queue q;  
    // no need to set  
    // q.nWaiting, q.front  
    // to 0.  
}
```

Constructors In General

```
struct A{  
    ...  
    A(parameters){  
        ...  
    }  
};  
  
int main(){  
    A a(arguments);  
}
```

- Constructor can take arguments
- The creation of the object **a** in main can be thought of as happening in two steps
 - Memory is allocated for **a** in main
 - The constructor is called on **a** with the given arguments
- You can have many constructors, provided they have different signatures

Another example: Constructor for V3

```
struct V3{  
    double x,y,z;  
    V3(){  
        x = y = z = 0;  
    }  
    V3(double a){  
        x = y = z = a;  
    }  
};  
int main();  
    V3 v1(5), v2;  
}
```

- When defining **v1**, an argument is given
- So **the constructor taking a single argument** is called. Thus each component of v1 is set to 5
- When defining v2, no argument is given. So **the constructor taking no arguments** gets called. Thus each component of v2 is set to 0

Remarks

- If and only if you do not define a constructor, will C++ defines a constructor for you which takes no arguments, and does nothing
 - If you define a constructor taking arguments, you implicitly tell C++ that you want programmers to give arguments. So if some programmer does not give arguments, C++ will flag it as an error
 - If you want both kinds of initialization, define both kinds of constructor
- A constructor that does not take arguments (defined by you or by C++) is called a default constructor
- If you define an array of struct, each element is initialized using the default constructor

The Copy Constructor

- Suppose an object is passed by value to a function
 - It must be copied to the variable denoted by the parameter
- Suppose an object is returned by a function
 - The value returned must be copied to a temporary variable in the calling program
- By default the copying operations are implemented by copying each member of one object to the corresponding member of the other object
 - You can change this default behaviour by defining a copy constructor

Example

```
struct Queue{
    int elements[N], nWaiting, front;
    Queue(const Queue &source){
        // Copy constructor
        front = source.front;
        nWaiting = source.nWaiting;
        for(int i=front, j=0; j<nWaiting; j++){
            elements[i] = source.elements[i];
            i = (i+1) % N;
        }
    };
};
```

Copy Constructor in the Example

- The copy constructor must take a single reference argument: the object which is to be copied
- *Note that the argument to the copy constructor must be a reference, otherwise the copy constructor will have to be called to copy the argument! This is will result in an unending recursion*
- Member elements is not copied fully. Only the useful part of it is copied
 - More efficient
- More interesting use later

Destructors

- When control goes out of a block in which a variable is defined, that variable is destroyed
 - Memory allocated for that variable is reclaimed
- You may define a destructor function, which will get executed before the memory is reclaimed

Destructor Example

- If a queue that you have defined goes out of scope, it will be destroyed
- If the queue contains elements at the time of destruction, it is likely an error
- So you may want to print a message warning the user
- It is usually an error to call the destructor explicitly. It will be called automatically when an object is to be destroyed. It should not get called twice.
- More interesting uses of the destructor will be considered in later chapters.

Destructor Example

```
struct Queue{
    int elements[N], nWaiting, front;
    ...
    ~Queue(){          //Destructor
        if(nWaiting>0)
            cout << "Warning:"
                <<" non-empty queue being destroyed."
                << endl;
    }
};
```

Operator Overloading

- In Mathematics, arithmetic operators are used with numbers, but also other objects such as vectors
- Something like this is also possible in C++!
- An expression such as `x @ y` where `@` is any “infix” operator is considered by C++ to be equivalent to `x.operator@(y)` in which `operator@` is a member function
- If the member function `operator@` is defined, then that is called to execute `x @ y`

Example: Arithmetic on V3 objects

```
struct V3{
    double x, y, z;
    V3(double a, double b, double c){
        x=a; y=b; z=c;
    }
    V3 operator + (const V3& b){ // adding two V3s
        return V3(x+b.x, y+b.y, z+b.z); // constructor
call
    }
    V3 operator * (double f){// multiplying a V3 by f
        return V3(x*f, y*f, z*f); // constructor call
    }
};
```

Using V3 Arithmetic

```
int main(){  
  
    V3 u(1,2,3), a(4,5,6), s;  
  
    double t=10;  
  
    s = u*t + a*t*t*0.5;  
  
    cout << s.x << ' ' << s.y << ' '  
        << s.z << endl;  
  
}
```

Remarks

- Expression involving vectors can be made to look very much like what you studied in Physics
- Other operators can also be overloaded, including unary operators (see the book)
- Overload operators only if they have a natural interpretation for the struct in question
- Otherwise you will confuse the reader of your program

The `this` pointer

- So far, we have not provided a way to refer to the receiver itself inside the definition of a member function.
- Within the body of a member function, the keyword `this` points to the receiver i.e. the struct on which the member function has been invoked.
- Trivial use: write `this->member` instead of `member` directly.

```
struct V3{  
    double x, y, z;  
    double length(){  
        return sqrt(*this.x * *this.x  
            + *this.y * *this.y  
            + *this.z * *this.z);  
    }  
}
```

- More interesting use later.

Overloading The Assignment Operator

- Normally if you assign one struct to another, each member of the rhs is copied to the corresponding member of the lhs
- You can change this behaviour by defining member function `operator=` for the struct
- A return type must be defined if you wish to allow chained assignments, i.e. `v1 = v2 = v3`; which means `v1 = (v2 = v3)`;
 - The operation must return a reference to the left hand side object

Example

```
struct Queue{
    ...
    Queue & operator=(Queue &rhs){
        front = rhs.front;
        nWaiting = rhs.nWaiting;
        for(int i=0; i<nWaiting; i++){
            elements[i] = rhs.elements[i];
            i = (i+1) % N;
        }
        return *this;
    }
};
// only the relevant elements are copied
```