The following materials have been collected from the numerous sources such as Stanford CS106 and Harvard CS50 including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

PSet: Map & Function pointer

Map containers

The STL(Standard Template Library) **maps** are **associative containers** that store elements formed by a combination of a **key** and a mapped **value**. Each key is unique and cannot be changed, and it can be inserted or deleted but cannot be altered. Value associated with key can be altered.
In an associative container (`map, unordered_map, set` etc), the items are not arranged in sequence, but usually as a **tree structure** or a **hash table** that we are going to learn later in this course. The main advantage of associative containers is the speed of searching (binary search like in a dictionary). Searching is done using a key which is usually a single value like a number or string.

Example

Let us suppose that we have a data set that consists of a set of strings and numbers. A map of men (a data set) where the name is the **key** and the age is the **value** can be listed and be able to access these data by its name:

Name	Ages
"John"	21
"Paul"	15
"Pete"	10
"Adam"	11

This table can be coded as follows:

```cpp
#include <iostream>
#include <map>
using namespace std;

int main() {
    map<string, int> table;
    cout << "using keys as array indices\n";
    table["John"] = 21;
    table["Paul"] = 15;
    table["Pete"] = 10;
    table["Adam"] = 11;

    cout << "using range-based for loop\n";
    for (auto x: table) {
        cout << "name: " << x.first << "\t";
        cout << " age: " << x.second << endl;
    }

    cout << "using iterator\n";
    for (auto it = table.begin(); it != table.end(); ++it) {
        cout << "name: " << it->first << "\t";
        cout << " age: " << it->second << endl;
    }
    return 0;
}
```

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset3sorting> ./map
using keys as array indices
using range-based for loop
name: Adam      age: 11
name: John      age: 21
name: Paul      age: 15
name: Pete      age: 10
using iterator
name: Adam      age: 11
name: John      age: 21
name: Paul      age: 15
name: Pete      age: 10
PS C:\GitHub\nowicx\psets\pset3sorting>
```

The table that consists of a list of **<name, age>** is initialized using keys as array indices. Then the table is printed in twice using both range-based for loop and iterator. As you observe the order of the keys, they are stored in a search tree structure. When we retrieved them, it returns elements in a sorted fashion.

There are two kinds of map in STL. If you care about the order of keys, you may use **map**, otherwise **unordered_map**. Using a **unordered_map**, it may produce as follows:

```
PS C:\GitHub\nowicx\psets\pset3sorting> g++ map.cpp -o map
PS C:\GitHub\nowicx\psets\pset3sorting> ./map
name: Adam      age: 11
name: Pete      age: 10
name: Paul      age: 15
name: John      age: 21
21
11
```

We can search, remove and insert in a **map** in **O(log n)** time complexity, and in **unordered map** in average O(1) and worst O(n) time complexity.

std::pair and std::make_pair()

`std::pair` is to combine two data into one data. The two data can be of the same type or different types. For example, `std::pair<int,float>` or `std::pair<double, double>`, etc. A `pair` is essentially a struct. The main two member variables are `first` and `second`.

To initialize a **`pair`**, you can use the constructor, or you can use the **`std::make_pair`** function. The **`make_pair`** function is defined as follows:

```
template pair make_pair(T1 a, T2 b) { return pair(a, b); }
```

The difference is that with **`std::pair`** you need to specify the types of both elements, whereas **`std::make_pair`** will create a pair with the type of the elements that are passed to it, without you needing to tell it.

For example, we may use **`pair`** construct or **`make_pair`** function to insert a map element as shown below:

```
12      table["John"] = 21;
13      table["Paul"] = 15;
14      table.insert(pair<string,int>("Pete",10));    // using insert() method
15      table.insert(make_pair("Adam",11));           // using pair<> or make_pair()
```

The following example shows the map initialization using **`pair`** and **`make_pair().`**

```
5   #include <iostream>
6   #include <map>
7   using namespace std;
8
9   int main() {
10      // using initialization, pair<> construct, and make_pair() function
11      map<char, int> chart {  pair<char,int>('A', 65),
12                              pair<char,int>('C', 67),
13                              make_pair('D', 68),
14                              make_pair('B', 66) };
15
16      for (auto item: chart) {
17          cout << "ascii: " << item.first << "\t";
18          cout << " code: " << item.second << endl;
19      }
20      cout << chart['B'] << endl;
21      return 0;
22  }
```

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset3sorting> g++ map.cpp -o map
PS C:\GitHub\nowicx\psets\pset3sorting> ./map
ascii: A         code: 65
ascii: B         code: 66
ascii: C         code: 67
ascii: D         code: 68
66
PS C:\GitHub\nowicx\psets\pset3sorting>
```

Observe that the chart was initialized with `'A', 'C', 'D'` and `'B'` initially, but it prints elements in a sorted way. It is a feature of `map` container. If you do not care about the sorting, you may use `unordered_map` or `set` container in STL.

Coding: map.cpp

While following instructions in skeleton code, write `map.cpp` such that it produces the output as shown **Sample Run** below:

```
#include <iostream>
#include <map>
using namespace std;

int main() {
    cout << "declare a map variable called table\n";
    map<string, int> table;

    cout << "initialize table using array[], insert(), pair<>, make_pair()\n";
    cout << "your code here\n";

    cout << "print table using range-based for loop\n";
    cout << "your code here\n";

    cout << "print table using iterator\n";
    cout << "your code here\n";

    cout << "define and initialize chart using pair<> and make_pair() only\n";
    cout << "your code here\n";

    cout << "print chart using range-based for loop\n";
    cout << "your code here\n";
    cout << "your code here\n";
    return 0;
}
```

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset3sorting> ./map
declare a map variable called table
initialize table using array[], insert(), pair<>, make_pair()
print table using range-based for loop
name: Adam      age: 11
name: John      age: 21
name: Paul      age: 15
name: Pete      age: 10
print table using iterator
name: Adam      age: 11
name: John      age: 21
name: Paul      age: 15
name: Pete      age: 10
define and initialize chart using pair<> and make_pair() only
print chart using range-based for loop
ascii: A        code: 65
ascii: B        code: 66
ascii: C        code: 67
ascii: D        code: 68
66
```

Function pointers

Function pointers provide some extremely interesting, efficient and elegant programming techniques. You can use them to replace switch/if-statements, to realize your own **late-binding** or to implement **callbacks.**

Unfortunately - probably due to their complicated syntax - they are treated quite carelessly in most computer books and documentations. If at all, they are addressed quite briefly and superficially. They are less error prone than normal pointers because you will never allocate or deallocate memory with them. All you've got to do is to understand what they are and to learn their syntax. But keep in mind: Always ask yourself if you really need a function pointer.

It's nice to realize one's own **late-binding** but to use the existing structures of C/C++ may make your code more readable and clearer.

- The function pointer tutorials - http://www.newty.de/fpt/intro.html#why
- 코딩도장 – 함수 포인터 만들기 https://dojang.io/mod/page/view.php?id=592

Definition

By definition, as you know, **pointers** point to an address in any memory location, they can also point to at the beginning of executable code as functions in memory.

Instead of referring to data values, a **function pointer** points to executable code within memory. When dereferenced, a function pointer can be used to invoke the function it points to and pass its arguments just like a normal function call.

A pointer to function is declared with the *, the general statement of its declaration is:

```
return_type (*function_name)(arguments)
```

You have to remember that the parentheses around `(*function_name)` are important because without them, the compiler will think the `function_name` is returning a pointer of `return_type.`

Function pointers can be used to simplify code by providing a simple way to select a function to execute based on run-time values. It is so called **late-binding.**

A function pointer always points to a function with a specific signature! Thus all functions, you want to use with the same function pointer, must have **the same parameters** and `return-type`!

Example

Let us make a simple code such that it tests the concept of function pointers. Take a following coding step by step.

1. Define a standard function, `greet()` which prints a **`"Hello World"`** text a number of times indicated by the parameter `times` when the function is called.
2. Declare a function pointer (with its special declaration) called `funptr` which takes an integer parameter `times` and doesn't return anything.
3. Initialize our function pointer `funptr` with the `greet` which means that the pointer points to the `greet().`
4. Instead of using the standard function call `greet(3),` let us use the function pointer `funptr` by passing the number 3 as arguments.

Recall that a pointer to function is declared with the * as shown below:

```
return_type (*funptr_name)(arguments)
```

You have to remember that the parentheses around `(*function_name)` are important because without them, the compiler will think the `function_name` is returning a pointer of `return_type.`

```
1   #include <iostream>
2   void greet(int times);
3
4   int main() {
5       void (*funptr) (int) = greet;
6       funptr(3);
7       return 0;
8   }
9
10  void greet(int times) {
11      for (int i = 0; i < times; i++)
12          std::cout << "Hello World" << std::endl;
13  }
```

```
PS C:\GitHub\nowicx\psets\pset3sorting> g++ fp1.cpp -o fp1
PS C:\GitHub\nowicx\psets\pset3sorting> ./fp1
Hello World
Hello World
Hello World
```

Keep in mind that the function name points to the beginning address of the executable code like an array name which points to its first element. Therefore, instructions like `funptr = &greet` and `(*funptr)(3)` are correct as well

Sample Run:

```
4   int main() {
5       void (*funptr) (int) = &greet;
6       (*funptr)(3);
7       return 0;
8   }
```

An array of function pointers

Differences from normal pointers:

- A function pointer points to code, not data. Typically, a function pointer stores the start of executable code.
- We do not allocate nor de-allocate memory using function pointers.

Same as normal pointers;

- We can have an array of function pointers.

```
int main() {
   // fps is an array of function pointers
   int (*fps[])(int, int) = { fun, foo, add };
   for (int i = 0; i < 3; i++)
      cout << "fps(" << i << ") returns " << fps[i](2, 3) << endl;;
}
```

A function pointer can be passed as an argument and can also be returned from a function. This feature of the function pointer is **extremely useful**. In OOP, class methods are another example implemented using function pointers.

Coding calc1.cpp

In this example, we're going to write a version of our basic calculator using function pointers.

Goals:

- NMN
- DRY
- Make it easy as possible when we add another operation such as mod % or power ^ operator.

Create a short program asking the user for two integer inputs and an arithmetical operation ('+', '-', '*', '/'). Ensure the user enters valid inputs and operation.

- Two functions, **`get_int()`** and **`get_op()`,** are provided.
- Four arithmetic functions, **`add()`, `sub()`, `mul()`,** and **`dvd()`,** are provided.
- Get familiar with the initialization coding style used at the beginning of **`main()`**. It is very common to use **`{}`** to initialize a variable during declaration.
- Complete the rest of the code as instructed below and as shown in Sample Run.
  - Declare a function pointer **`(*fp)`** for arithmetic operation.
  - Use **`switch()`** that use **`op`** to branch each arithmetic operation.
  - Print the result along with the arithmetic expression.

```
// 2021/02/15 created by idebtor@gmail.com
#include <iostream>
#include <sstream>
using namespace std;

int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }
int sub(int a, int b) { return a - b; }
int dvd(int a, int b) { if (b != 0) return a / b; else return 0; }

int get_int() {···

char get_op(string opstr) {···

int main() {
    int a { get_int() };                    // initialize a with user's input
    char op { get_op("+-*/") };             // get an operator chosen by user
    int b { get_int() };                    // initialize b with user's input

    int (                                   // declare a function pointer
    swit
        default: break;
    }
    cout                                    << endl;
    return 0;
}
```

```
int get_int() {
    int x;
    do {
        cout << "Enter an integer: ";
        string str;
        getline(cin, str);
        try {
            x = stoi(str);
            break;
        }
        catch (invalid_argument& e) {
            cerr << e.what() << " error occurred. Retry~" << endl;
        }
    } while(true);
    return x;
}
```

```
char get_op(string opstr) {
    char op;
    do {
        stringstream ss;
        string str;
        cout << "Enter an operator( " << opstr << " ): ";
        getline(cin, str);
        ss << str;
        ss >> op;                           // find() returns npos if not found
    } while (opstr.find(op) == string::npos);   // find() returns index op in opstr
    return op;
}
```

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset3sorting> g++ -Wall calc1.cpp -o calc1
PS C:\GitHub\nowicx\psets\pset3sorting> ./calc1
Enter an integer: addtion
stoi error occurred. Retry~
Enter an integer: 11
Enter an operator( +-*/ ): plus
Enter an operator( +-*/ ): +
Enter an integer: 33
11 + 33 = 44
PS C:\GitHub\nowicx\psets\pset3sorting>
```

As you observe, `get_int()` and `get_op()`check the validity of user's input.

Coding calc2.cpp

Just like other variables, we may create an array of function pointers. Then, an array of function pointers can play a `switch` or an `if` statement role for making a decision. In this program, we do not want to use a switch statement, **while using an array of function pointers**:

1. To replace the switch statement, we define an array of function pointers and use its index to access a particular arithmetic function.
2. Modify `get_op(string opstr)` function such that it returns operator and its index in `opstr` argument. In C/C++, however, the function can return only one value or pointer. To return both operator char and its index in opstr, we may store and return them in a `pair<char, int>` object. The `get_op()` becomes as follows:

```
28  pair<char,int> get_op(string opstr) {
29      char op;                                    // user's operator entered
30      size_t x;                                   // index of op in opstr
31      do {
32          cout << "Enter an operator( " << opstr << " ): ";
33          stringstream ss;
34          string str;
35          getline(cin, str);
36          ss << str;
37          ss >> op;
38          x =                                     // find index of op in opstr
39      } while                                     // while op is not found in opstr
40      return m                                    // returns an operator and its index
41  }
```

3. Now, in the `main()`, we use the operator and its index stored in `pair<char,int>` structure to perform the arithmetic computation and print the result.

```
43  int main() {
44      int                                         // array of function pointers
45      string opstr { "+-*/" };                    // operators in string
46
47      int a { get_int() };                        // initialize a with user's input
48      pair<char,int> op { get_op(opstr) };        // get an operator and its index
49      int b { get_int() };                        // initialize b with user's input
50
51      cout <<                                     << endl;
52      return 0;
53  }
54
```

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset3sorting> g++ -Wall calc2.cpp -o calc2
PS C:\GitHub\nowicx\psets\pset3sorting> ./calc2
Enter an integer: 33
Enter an operator( +-*/ ): *
Enter an integer: 2
33 * 2 = 66
PS C:\GitHub\nowicx\psets\pset3sorting>
```

Coding calc3.cpp

In the previous example, we have two lists. The one is an array of function pointers such as **`{ add, sub, mul, dvd }`** and the other is a matching sequence of operators, **`"+-*/"`**, corresponding to the list of function pointers. It is not a good idea of maintaining two sequences for one operation. Then, it becomes error – prone code.

Let us use a **`map`** container that maps an operator to a function pointer. For example, the key **`'+'`** is associated with its function pointer of **`add()`**. In code,

```
fp_map['+'] = add;        // declare map first and then assign fp.
fp_map['-'] = sub;        // it is ok, but do not use in calc3.cpp
```

1. Instead of using an array of function pointer, **define** a map in STL that maps four arithmetic operators **`('+', '-', '*', '/')`** to four arithmetic functions **`(add, sub, mul, dvd)`**, respectively and initialize them using **`make_pair()`.**
   During defining the map, use **`make_pair()`** function instead of using an array notation. For example, **`make_pair('+', add)`**

```
46  int main() {
47      map<char,int(*)
48
49
50      int a { get_int() };                        // initialize a with user's input
51      char op { get_op(fp_map) };                 // get an operator and its index
52      int b { get_int() };                        // initialize b with user's input
53
54      cout <<                                     << endl;
55      return 0;
56  }
```

2. Modify `get_op()`such that it takes a map container `map<char,int(*)(int,int)> fp_map` and returns an operator `char`.

```
29  char get_op(map<char,int(*)(int,int)> fp_map) {
30      string opstr;
31      char op;
32      for (auto x: fp_map) opstr += x.first;      // create opstr from fp_map's keys
33
34      do {
35          cout << "Enter an operator( " << opstr << " ): ";
36          stringstream ss;
37          string str;
38          getline(cin, str);
39          ss << str;
40          ss >> op;                       // if not found, find() returns fp_map.end()
41          if (fp_map.                     break;   // if found, op is valid.
42      } while (true);                     // while op is not found in fp_map
43      return op;                          // returns the operator chosen by user
44  }
```

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset3sorting> g++ -Wall calc3.cpp -o calc3
PS C:\GitHub\nowicx\psets\pset3sorting> ./calc3
Enter an integer: 123
Enter an operator( *+-/ ): +
Enter an integer: 321
123 + 321 = 444
PS C:\GitHub\nowicx\psets\pset3sorting>
```

Congratulations!

Submitting your solution

- Include the following line at the top of your every source file with your name signed.
  On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  Signed: _____ **Student Number:** _____
- Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not get sympathy for code that "almost" works.
- If you only manage to work out the Project problem partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit, Due and Grade points

Files to submit: upload the following files in piazza folder

- map.cpp, calc1.cpp, calc2.cpp, calc3.cpp

Due: 11:55 pm