본 PSet 은 저의 강의 경험과 학생들의 의견 및 Stanford CS106 과 Harvard CS50 같은 강의에서 수집된 자료를 토대로 작성되었습니다.
본 PSet 에 문제점이나 질문 혹은 의견이 있다면, 저의 이메일(idebtor@gmail.com)로 알려 주시면 강의 개선에 많은 도움이 되겠습니다.

PSet: Map & Function pointer

Map containers

C++ STL(Standard Template Library)에서 제공하는 **map** 데이터 타입은 **key** 와 매핑된 **value** 의 조합으로 형성된 요소를 저장하는 연관 컨테이너(associative containter)입니다. 맵의 각 키는 고유하며 변경할 수 없으며 삽입 또는 삭제는 가능하지만 수정할 수 없습니다. 키와 관련된 값은 변경될 수 있습니다.
연관 컨테이너(map, unordered_map, set 등)에서 항목은 순차적으로 배열되지 않고 이 강의 후반부에서 배울 **tree structure** 나 **hash table** 와 같이 배열됩니다.
연관 컨테이너의 주요 장점은 검색 속도입니다(dictionary 처럼 이진 검색). 검색은 일반적으로 숫자나 문자열과 같은 단일 값인 키를 사용하여 수행할 수 있습니다.

예시

문자열과 숫자의 집합으로 구성된 데이터 집합이 있다고 가정합시다. name 이 **key** 이고 ages 가 **value** 인 남성을 조사한 map(데이터 세트)을 나열할 수 있으며 name 을 통해 데이터에 접근할 수 있습니다:

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

이 표는 다음과 같이 코딩 합니다:

```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;
}
```

실행 예시:

```
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>
```

<name, age> 목록으로 구성된 테이블은 key 를 배열 인덱스로 사용하여 생성됩니다. 그런 다음 표는 각각 ranged-for-문과 iterator(반복자)를 사용하여 두 번 인쇄됩니다. key 의 순서를 관찰하면 key 가 search tree(검색 트리) 구조로 저장되는 것을 알 수 있습니다. 검색하면 요소들이 정렬된 방식으로 반환됩니다.

STL 에는 두 가지 종류의 map 이 있습니다. key 의 순서를 신경 쓴다면 **map** 을 사용할 수도 있고, 그렇지 않으면 **unordered_map** 을 사용할 수도 있습니다. **unordered_map** 을 사용하면 다음과 같이 코딩 합니다:

```
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
```

map 에서 검색, 제거, 삽입은 **O(log n)**시간 복잡도를 가집니다. 그리고 **unordered map** 에서는 평균적으로 **O(1)**이지만 최악의 경우 **O(n)**시간 복잡도를 가집니다.

std::pair 와 std::make_pair()

std::pair 는 두 데이터를 하나의 데이터로 결합하는 것입니다. 두 데이터는 동일한 타입일 수도 있고 다른 타입일 수도 있습니다. 예를 들어 std::pair<int,float> 또는 std::pair<double,double> 등이 있습니다. 한 쌍은 본질적으로 데이터 구조입니다. 주가 되는 두 멤버 변수는 **first** 와 **second** 입니다.

pair 를 생성하려면 생성자를 사용하거나 **std::make_pair** 함수를 사용할 수 있습니다. **make_pair** 함수는 다음과 같이 정의됩니다:

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

그 둘의(std::pair, std::make_pair) 차이점은: **std::pair** 의 경우 두 요소의 타입을 명시해야 하는 반면 **std::make_pair** 는 명시할 필요없이 보내는 요소의 타입에 따라 자동으로 pair 를 생성합니다.

예를 들어, 다음과 같이 **pair** 생성자나 **make_pair** 함수를 사용하여 map 요소를 삽입할 수 있습니다:

```
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()
```

다음 예시는 **pair** 와 **make_pair()**를 이용한 map 생성한 것 입니다:

```
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  }
```

실행 예시:

```
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>
```

결과를 관찰해보면, 차트는 처음에 'A', 'C', 'D' 및 'B'로 생성되었지만 정렬된 방식으로 요소들을 출력합니다. 이것은 **map** container 의 특징입니다. 정렬에 상관없는 경우 **unordered_map** 을 사용하거나 STL 에서 **set** container 를 사용할 수 있습니다.

코딩: map.cpp

Skeleton code 의 지침을 따르면서, 아래에 표시된 **실행 예시**와 같이 출력이 생성되도록 **map.cpp** 파일을 작성하십시오:

```
#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;
}
```

실행 예시:

```
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 pointer(함수 포인터)는 매우 흥미롭고 효율적이며 우아한 프로그래밍 기술을 제공합니다. 함수 포인터를 사용하여 switch/if-문을 대체하거나, 자신의 **late-binding**(후기 바인딩)을 실현하거나, **callback**(콜백함수)을 구현할 수 있습니다.

- Late-binding : https://www.educative.io/answers/what-are-early-binding-and-late-binding-functions-in-cpp
- Callback: https://stackoverflow.com/questions/2298242/callback-functions-in-c#:~:text=A%20callback%20is%20a%20callable,be%20reused%20with%20different%20callbacks.

하지만 복잡한 구문을 가지고 있는 탓에, 대부분의 컴퓨터 서적과 문서들은 그것을 잘 다루지 않습니다. 만약 설명하는 책들이 있어도, 대부분 꽤 짧고 피상적으로 다루어집니다. 함수 포인터는 일반 포인터보다 메모리를 할당하거나 할당 해제하지 않기 때문에 그렇게 어려운 것도 아닙니다.

여러분이 해야 할 일은 그것들이 무엇인지 이해하고 그들을 잘 활용할 수 있는 기법을 배우는 것입니다. **하지만 명심하세요**: 함수 포인터가 정말 필요한지 항상 스스로 질문해 보십시오. 왜냐하면, **late-binding** 을 실현하는 것은 좋지만 C/C++의 기존/존재하는 구조를 사용함으로 코드를 더 읽기 쉽고 명확하게 만들 수도 있기 때문입니다.

- 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

정의

정의에 따르면, 이미 배우셨겠지만, **포인터**는 메모리 위치의 주소를 가리키며, 실행 코드의 시작 부분을 메모리의 함수로 가리킬 수도 있습니다.

데이터 값을 참조하는 대신 **함수 포인터는** 메모리 내의 실행 가능 코드를 가리킵니다. 참조가 해제되면 **함수 포인터**를 사용하여 일반 함수 호출처럼 해당 함수를 호출하고 인수를 전달할 수 있습니다.

함수에 대한 포인터는 *로 선언되며, 그 선언의 일반적인 문장은 다음과 같습니다:

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

(*function_name), 겉에 괄호를 추가하는 것을 잊으면 안됩니다. 괄호가 없으면 컴파일러가 function_name 이 return_type 를 가리키는 포인터를 반환한다고 생각합니다.

함수 포인터를 사용하면 손쉽게 실행 하는 동안 생성되는 값들의 의하여 실행결과가 유동성 있게 해줍니다. 이 것을 **late-binding** 이라고도 부릅니다.

함수 포인터는 항상 특정한 함수 고유의 값을 가리킵니다! 그렇기 때문에, 함수 포인터를 사용하고 싶은 모든 함수들은 함수 포인터와 같은 **매개변수**와 **반환 타입**를 가지고 있어야 합니다.

예시

간단한 코드를 작성하여 지금까지 배운 함수 포인터를 실험해 봅시다. 다음 코딩 단계들을 step by step 따라하면 됩니다.

1. greet() 함수를 정의합시다. greet() 함수는 **"Hello World"**를 함수가 호출될 때 매개변수를 통해 받은 수만큼 출력합니다.
2. 정수를 매개변수로 받고, 아무것도 반환하지 않는 funptr 라고 불리는 함수 포인터를 하나 정의합니다(위에서 배운 방법으로).
3. 위 단계에서 생성한 함수 포인터(funptr)를 greet() 함수로 초기화 합니다. 이는 이제 funptr 는 greet()를 가리키게 됩니다.
4. 기존 함수 호출 방법인 greet(3)이 아니라, funptr 에 3 을 변수로 보내서 함수 포인터를 사용합니다.

앞서 배웠던 *을 사용해 함수를 가리키는 포인터를 선언하는 방법입니다~^^:

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

(*funptr_name), 겉에 괄호를 추가하는 것 잊지 않으셨죠? 괄호가 없으면 컴파일러가 funptr_name 이 return_type 를 가리키는 포인터를 반환한다고 생각합니다. 꼭 기억해야 합니다.

```cpp
#include <iostream>
void greet(int times);

int main() {
    void (*funptr) (int) = greet;
    funptr(3);
    return 0;
}

void greet(int times) {
    for (int i = 0; i < times; i++)
        std::cout << "Hello World" << std::endl;
}
```

실행 예시:

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

함수의 이름은 함수를 실행하는 코드의 메모리 시작 주소를 가리킵니다. 배열의 이름이 배열의 첫번째 값이 들어있는 주소를 가리키는 것과 같은 개념입니다. 그렇기 때문에, funptr = &greet 와 (*funptr)(3) 같은 명령들도 정상적으로 작동합니다.

실행 예시:

```cpp
int main() {
    void (*funptr) (int) = &greet;
    (*funptr)(3);
    return 0;
}
```

함수 포인터 배열

일반 포인터와 함수 포인터의 **차이점**:

- 함수 포인터는 값이나 데이터가 아니라 코드를 가리킵니다. 일반적으로 함수 포인터는 실행되는 코드의 시작을 저장하고 있습니다.
- 함수 포인터를 사용할 때는 메모리를 할당하거나 할당 해제 하지 않습니다.

일반 포인터와 함수 포인터의 **공통점**:

- 함수 포인터로 구성된 배열을 생성 할 수 있습니다.

```
int main() {
   // fps 는  함수 포인터로 이루어진 배열입니다.
    int (*fps[])(int, int) = { fun, foo, add };
    for (int i = 0; i < 3; i++)
      cout << "fps(" << i << ") returns " << fps[i](2, 3) << endl;;
}
```

함수 포인터는 매개변수로 전달되는 변수가 될 수 있고 함수에서 반환될 수도 있습니다. 이 점은 함수 포인터를 **아주 유용하게** 만들어 줍니다. OOP(객체지향적 프로그래밍)의 class(클래스)에 정의되어 있는 멤버 함수들은 함수 포인터 형식으로 저장된 함수들입니다. 함수 포인터를 사용하는 좋은 예시입니다.

코딩: calc1.cpp

이 예시에서는 함수 포인터를 사용하여 우리만의 기본적인 계산기를 구현할 것입니다.

목표:

- NMN
- DRY
- 나머지 %또는 제곱 ^, 연산자와 같은 다른 연산자를 추가하기 최대한 쉽도록 만듭니다.

먼저, 사용자에게 정수 두개와 연산자('+', '-', '*', '/')를 입력 받는 간단한 프로그램을 작성합니다. 유효한 숫자 또는 연산자를 입력했는지 검사도 합니다.

- **get_int()**와 **get_op()** 두 함수는 제공 됩니다.
- **`add()`, `sub()`, `mul()`, `dvd()`**, 네 개의 연산 함수도 제공 됩니다.
- main()의 시작 부분에 사용되는 initialization(초기화) 코딩 방식을 숙지하십시오. 선언 중에 변수를 초기화하는 데 {}을(를) 사용하는 것은 매우 일반적입니다.
- 밑 단계들을 따라 남은 코드를 작성하고, 실행 예시와 같이 작동시키면 됩니다.
  - 연산에 사용될 **(*fp)** 함수 포인터를 선언합니다.
  - **switch()**를 사용해 **op** 을 각 연산과정으로 분류되도록 합니다.
  - 결과와 함께 연산식도 출력합니다.

```cpp
// 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;
}
```

```cpp
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;
}
```

```cpp
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;
}
```

실행 예시:

```
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>
```

위 코드에 나타나듯이 get_int()와 get_op()는 user(사용자)의 input(입력)이 유효한지 확인합니다.

코딩: calc2.cpp

다른 변수들과 같이 함수 포인터를 배열로 만들 수 있습니다. 배열로 이루어진 함수 포인터들이 있으면 switch 또는 if-문을 이용해 결정하는데 도움을 줄 수 있습니다. 이 프로그램 단계에서 **함수 포인터로 이루어진 배열을 사용할 때는** switch 문을 사용하지 않습니다.

1. Switch 문을 교체하기 위해서 함수 포인터로 이루어진 배열의 index(인덱스)를 사용하여 특정한 연산 함수를 접근 할 수 있도록 배열을 선언합니다.
2. **get_op(string opstr)** 함수를 수정하여 연산자와 그 연산자의 인덱스를 반환하도록 합니다. 하지만, C/C++에 함수는 한 값이나 포인터만 반환할 수 있습니다. 두 가지(연산자 char, index **opstr**) 모두 return 하기 위해서는 **pair<char, int>**에 저장하여 반환할 수 있습니다. 이러한 논리 이용한 **get_op()**는 다음과 같습니다:

```
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. main()에서는, **pair<char, int>**에 저장 되어있는 연산자와 인덱스를 이용해 연산과정을 거쳐 결과를 출력합니다.

```
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
```

실행 예시:

```
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>
```

코딩: calc3.cpp

이전 예제에서는 두 개의 리스트가 있었습니다. 하나는 **{add, sub, mul, dvd}**와 같은 함수 포인터의 배열이고, 다른 하나는 함수 포인터의 목록에 해당하는 연산자 **"+-*/"**의 일치 순서이었습니다. 하나의 작업에 대해 두 개의 시퀀스를 유지하는 것은 좋은 생각이 아닙니다. 그러면 오류가 발생하기 쉬운 코드가 됩니다.

연산자를 함수 포인터에 매핑하는 **map** 컨테이너를 사용합시다. 예를 들어 key **'+'**는 함수 포인터인 **add()**와 연결되어 있습니다. 코드에서는:

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

1. 함수 포인터의 배열을 사용하는 대신 STL 에서 4 개의 산술 연산자**('+', '-', '*', '/')**를 각각 4 개의 산술 함수**(add, sub, mul, dvd)**에 매핑하는 map 을 **정의**하고 **make_pair()**를 사용하여 초기화합니다.

   map 을 정의하는 동안 배열 표기법 대신 **make_pair()** 함수를 사용합니다. 예를 들어 **make_pair('+', add)**와 같이 입력합니다.

```
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. map 컨테이너인 map **<char,int(*)> fp_map** 을 가져와서 연산자 **char** 를 반환하도록 **get_op()**을 수정합니다.

```
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  }
```

실행 예시:

```
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>
```

축하합니다!

과제 제출

- 소스 파일 상단에 아래와 같이 아너 코드 문장을 적고 서명하세요.
  On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  서명: _____ 분반: _____ 학번: _____
- 제출하기 전에 코드가 제대로 컴파일이 되고 실행되는지 확인하세요. 제출 직전에 급하게 코드를 수정한 후 코드가 제대로 컴파일이 될 거라고 짐작하지 않는 게 좋습니다. “거의” 작동하는 코드도 틀린 것입니다.

- 과제가 컴파일 및 실행된다면, 마감 기한 전까지 과제의 일부만 완성했더라도 제출하기 바랍니다. 컴파일 및 실행되지 않는다면 제출하지 마세요. 마감 시간 이후 24 시간 이내 제출하면, 만점에서 25% 감점하고 채점합니다. 그 이상 늦은 것은 채점하지 않으며, 0 점 처리합니다.
- 제출 후, **마감 기한 전까지** 수정 및 재제출이 가능합니다. 파일 하나만 수정하더라도 해당 파일과 관련된 파일들을 모두 재제출해야 합니다. 재제출 횟수는 제한 없습니다. 마감 기한 전에 **가장 마지막으로** 제출된 파일을 채점할 것입니다.
-

제출 파일 목록, 마감기한 & 배점

다음 파일들을 piazza 폴더에 제출하세요.

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

마감기한: 11:55 pm