Chapter 7

Function parameters and arguments

Passing arguments by value (call by value)

```
#include <iostream>

void foo(int y)
{
    std::cout << "y = " << y << '\n';
}

int main()
{
    foo(5); // first call by value

    int x = 6;
    foo(x); // second call
    foo(x+1); // third call

    return 0;
}
```

> Pros and cons of pass by value

Advantages of passing by value:

- Argument로 될 수 있는 값은 변수, 리터럴, 익스프레션, 구조체와 클래스, 이뉴머레이터 즉, 뭐든 다 돼요.
- Argument는 함수 호출로 인해서 절대 변하지 않는다. 즉 사이드 이펙트가 없어요.

Disadvantages of passing by value:

- 구조체와 클래스의 복제가 일어나요. 함수가 매우 많이 호출되면 엄청난 성능하락을 야기해요.

When to use pass by value:

- 기본 데이터 타입이나 이뉴머레이터가 아규먼트인 경우이거나 아규먼트가 함수 호출로 인해서 변하지 않는 경우 사용하세요.

When not to use pass by value:

- 구조체나 클래스가 아규먼트인 경우 (각종 컨테이너나 배열, 스트링을 포함) 사용하지 마세요.

Passing arguments by reference (call by reference)

```
void addOne(int &ref)
{
    ref = ref + 1;
}

int main()
{
    int value = 5;

    cout << "value = " << value << '\n';
    addOne(value);
    cout << "value = " << value << '\n';
    return 0;
}
```

> Pros and cons of pass by reference

Advantages of passing by reference:

- 레퍼런스 파라미터는 호출한 함수가 아규먼트를 변경할 수 있게 해줍니다. 그렇지만, `const` 레퍼런스 파라미터는 호출한 함수가 아규먼트를 변경하지 않는다는 것을 보장하기도 합니다.
- 아규먼트의 복사가 이루어지지 않기 때문에 큰 자료형을 넣어도 빠릅니다.
- 함수가 파라미터를 통해서 반환 할 수 있게 합니다.
- 레퍼런스는 사용하기 전에 항상 초기화가 의무인 값이기 때문에 널 값 오류를 범하지 않습니다.

Disadvantages of passing by reference:

- `non-const` 레퍼런스 파라미터로 쓰일 아규먼트는 `const l-value` 혹은 `r-value` (리터럴 혹은 익스프레션) 일 수 없습니다.
- 함수의 호출과 아규먼트만 봐서는 이것이 인풋 파라미터인지 아웃풋 파라미터인지 혹은 함수 안에서 변하는 값일지 유추할 수 없습니다. (의식적으로 `const` 변수를 활용하거나, 아웃풋 변수의 경우 이름에 접미사를 붙이는 것이 도움될 수 있습니다.)

When to use pass by reference:

- 구조체나 클래스를 패스할 경우 (`read-only`일 경우 반드시 `const` 키워드 사용하기)
- 함수가 아규먼트를 수정하는 경우

When not to use pass by reference:

- 수정될 일 없는 기본 데이터 타입을 패스할 경우 (이때는 그냥 pass by value 쓰세요)

Passing arguments by address (call by address)

```
#include <iostream>

void foo(int *temp_ptr);
void printArray(int *array, int length);

int main()
{
    int value = 5;
    int *ptr = &value;
    std::cout << "value = " << value << '\n';
    foo(ptr);
    std::cout << "value = " << value << '\n';

    int array[6] = { 6, 5, 4, 3, 2, 1 };
    printArray(array, 6);

    return 0;
}

void foo(int *temp_ptr)
{
    *temp_ptr = 6;
}

void printArray(int *array, int length)
{
    // if user passed in a null pointer for array, bail out early!
    if (!array)
        return;

    for (int index=0; index < length; ++index)
        cout << array[index] << ' ';
}
```

> Pros and cons of pass by address

Advantages of passing by address:

- 아규먼트는 주소이지만 포인터가 가리키는 값을 함수가 바꿀 수 있게 해준다.
- 아규먼트가 포인터이므로 포인터가 가리키는 값은 복사가 일어나지 않아서 빠르다.
- 다중 반환을 파라미터를 통해 가능하게한다.

Disadvantages of passing by address:

- 리터럴과 표현식은 주소가 없기 때문에 아규먼트로 사용할 수 없다.
- 포인터의 널 체크를 반드시 해야 나중에 크러쉬가 발생하지 않는다. (포인터가 비어있을 때 디레퍼런싱을 시도하면 프로그램이 충돌한다 그래서 포인터는 주소가 가리키는 값을 사용하기 전에 항상 비어있는지 검사해야한다.)

When to use pass by address:

- 배열을 넘길 경우 (항상 크기와 함께 넘기세요.)
- 널포인터일 수도 있는 포인터를 넘길 경우

When not to use pass by address:

- 널포인터가 절대로 발생할 일 없는 포인터인 경우 (패스 바이 레퍼런스를 쓰세요.)
- 구조체 혹은 클래스 (패바레)
- 기본타입 (패스 바이 벨류)

Returning values by value, reference, and address

Write function prototypes for each of the following functions. Use the most appropriate parameter and return types (by value, by address, or by reference), including use of const where appropriate.

1. A function named SumTo() that takes an integer parameter and returns the sum of all the numbers between 1 and the input number.
2. A function named PrintEmployeeName() that takes an Employee struct as input.
3. A function named MinMax() that takes two integers as input and passes back to the caller the smaller and larger number as separate parameters.
4. A function named GetIndexOfLargestValue() that takes an integer array (as a pointer) and an array size, and returns the index of the largest element in the array.
5. A function named GetElement() that takes an integer array (as a pointer) and an index and returns the array element at that index (not a copy). Assume the index is valid, and the return value is const.

Function overloading

```
int Add(int x, int y); // integer version
double Add(double x, double y); // floating point version

int Add(int x, int y) {
    return x + y;
}

double Add(double x, double y) {
    return x + y;
}
```

Function return types are not considered for uniqueness

```
template<typename T>
T Add(T x, T y) {
    return x + y;
}
```

Default parameters

```
void printValues(int x, int y=10);

int main() {
    printValues(1); // y will use default parameter of 10
    printValues(3, 4); // y will use user-supplied value 4
}
```

```
void printValues(int x, int y) {
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

Calling a fuction using a fuction pointer

```
int foo(int x) {
    return x;
}

int main() {
    int (*fcnPtr)(int) = foo; // assign fcnPtr to function foo
    fcnPtr(5); // call function foo(5) through fcnPtr.

    return 0;
}
```

```
#include <algorithm> // for std::swap, use <utility> instead if C++11
#include <iostream>

// Note our user-defined comparison is the third parameter
void SelectionSort(int *array, int size, bool (*comparison_fn)(int, int));
// Here is a comparison function that sorts in ascending order
// (Note: it's exactly the same as the previous ascending() function)
bool Ascending(int x, int y);
// Here is a comparison function that sorts in descending order
bool Descending(int x, int y);
// This function prints out the values in the array
void PrintArray(int *array, int size);

int main()
{
    int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    // Sort the array in descending order using the descending() function
    SelectionSort(array, 9, Descending);
    PrintArray(array, 9);

    // Sort the array in ascending order using the ascending() function
    SelectionSort(array, 9, Dscending);
    PrintArray(array, 9);

    return 0;
}

void SelectionSort(int *array, int size, bool (*comparison_fn)(int, int))
{
    // Step through each element of the array
    for (int start_idx = 0; start_idx < size; ++start_idx)
    {
        // best_idx is the index of the smallest/largest element we've encountered
so far.
        int best_idx = start_idx;

        // Look for smallest/largest element remaining in the array (starting at
start_idx+1)
        for (int current_idx = start_idx + 1; current_idx < size; ++current_idx)
        {
            // If the current element is smaller/larger than our previously found
smallest
            if (comparison_fn(array[best_idx], array[current_idx])) // COMPARISON
DONE HERE
                // This is the new smallest/largest number for this iteration
                best_idx = current_idx;
        }

        // Swap our start element with our smallest/largest element
        std::swap(array[start_idx], array[best_idx]);
    }
}

bool Ascending(int x, int y)
{
    return x > y; // swap if the first element is greater than the second
}

bool Descending(int x, int y)
{
    return x < y; // swap if the second element is greater than the first
}

void PrintArray(int *array, int size)
{
    for (int index=0; index < size; ++index)
        std::cout << array[index] << " ";
    std::cout << '\n';
}
```

Comprehensive quiz

Q1: What’s wrong with these programs?

> a)

```
int& doSomething()
{
    int array[] = { 1, 2, 3, 4, 5 };
    return array[3];
```

~~~
}
```

> b)

```cpp
int sumTo(int value)
{
    return value + sumTo(value - 1);
}
~~~

> c)

```cpp
float divide(float x, float y)
{
    return x / y;
}

double divide(float x, float y)
{
    return x / y;
}
```

> d)

```cpp
#include <iostream>

int main()
{
    int array[100000000];

    for (const auto &x: array)
        std::cout << x << ' ';

    return 0;
}
```

> e)

```cpp
#include <iostream>

int main(int argc, char *argv[])
{
    int age = argv[1];
    std::cout << "The users age is " << age << '\n';
```

```
    return 0;
}
```

Q2: Write an iterative version of the binarySearch function.

The best algorithm for determining whether a value exists in a sorted array is called binary search.

Binary search works as follows:

- Look at the center element of the array (if the array has an even number of elements, round down).
- If the center element is greater than the target element, discard the top half of the array (or recurse on the bottom half)
- If the center element is less than the target element, discard the bottom half of the array (or recurse on the top half).
- If the center element equals the target element, return the index of the center element.
- If you discard the entire array without finding the target element, return a sentinel that represents “not found” (in this case, we’ll use -1, since it’s an invalid array index).

Because we can throw out half of the array with each iteration, this algorithm is very fast. Even with an array of a million elements, it only takes at most 20 iterations to determine whether a value exists in the array or not! However, it only works on sorted arrays.

Modifying an array (e.g. discarding half the elements in an array) is expensive, so typically we do not modify the array. Instead, we use two integer (min and max) to hold the indices of the minimum and maximum elements of the array that we’re interested in examining.

Let’s look at a sample of how this algorithm works, given an array `{ 3, 6, 7, 9, 12, 15, 18, 21, 24 }`, and a target value of `7`. At first, `min = 0`, `max = 8`, because we’re searching the whole array (the array is `length 9`, so the index of the last element is `8`).

- Pass 1) We calculate the midpoint of min (0) and max (8), which is 4. Element #4 has value 12, which is larger than our target value. Because the array is sorted, we know that all elements with index equal to or greater than the midpoint (4) must be too large. So we leave min alone, and set max to 3.
- Pass 2) We calculate the midpoint of min (0) and max (3), which is 1. Element #1 has value 6, which is smaller than our target value. Because the array is sorted, we know that all elements with index equal to or lesser than the midpoint (1) must be too small. So we set min to 2, and leave max alone.
- Pass 3) We calculate the midpoint of min (2) and max (3), which is 2. Element #2 has value 7, which is our target value. So we return 2.

Hint: You can safely say the target element doesn’t exist when the min index is greater than the max index.

```
#include <iostream>
#include <cassert>
// array is the array to search over.
// target is the value we're trying to determine exists or not.
// min is the index of the lower bounds of the array we're searching.
// max is the index of the upper bounds of the array we're searching.
// BinarySearch() should return the index of the target element
//                if the target is found, -1 otherwise.
```

```cpp
int BinarySearch(const int *array, int target, int min_idx, int max_idx);

int main() {
  const int ARRAY_MAX_SIZE = 15;
  const int array[ARRAY_MAX_SIZE] = {
     3,  6,  8, 12, 14,
    17, 20, 21, 26, 32,
    36, 37, 42, 44, 48
  };

  // We will test a bunch of values to see if they produce the expected results
  const int TEST_SIZE = 9;
  // Here are the test target values
  const int target_values[TEST_SIZE] = {0, 3, 12, 13, 22, 26, 43, 44, 49};
  // And here are the expected results for each value
  const int expected_values[TEST_SIZE] = {-1, 0, 3, -1, -1, 8, -1, 13, -1};

  // Loop through all of the test values
  for (int count = 0; count < TEST_SIZE; ++count) {
    // See if our test value is in the array
    int min_idx = 0;
    int max_idx = ARRAY_MAX_SIZE - 1;
    int index = BinarySearch(array, target_values[count], min_idx, max_idx);
    // If it matches our expected value, then great!
    if (index == expected_values[count]) {
      std::cout << "test value "     << target_values[count]   << " passed!\n";
      std::cout << "expected value " << expected_values[count] << "\n";
      std::cout << "index "          << index                  << "\n";
    }
    else  // otherwise, our BinarySearch() function must be broken
      std::cout << "test value " << target_values[count]
                << " failed.  There's something wrong with your code!\n";
  }

  return EXIT_SUCCESS;
}

int BinarySearch(const int *array, int target, int min, int max) {
}
```