



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 - Binary Search & Template

목차

Getting Started - 제공되는 파일 목록	1
C++ 함수 템플릿	1
함수 템플릿이란?	2
C++에서 함수 템플릿 생성하기	2
함수 템플릿 사용하기	3
이진 탐색에 대하여	4
Step 1: binsearch.cpp 구현하기	4
알고리즘 테스트하기	6
Step 2: binsearchT.cpp - 템플릿 버전	6
과제 제출	7
제출 파일 목록 & 배점	7
마감 기한	8

Getting Started - 제공되는 파일 목록

binsearch.cpp - 뼈대 코드

binsearchx, binsearchx.exe - Mac 과 Windows 용 실행 파일

binsearchTx, binsearchTx.exe - Mac 과 Windows 용 실행 파일

C++ 함수 템플릿

지금까지 프로그램을 더 안전하고, 유지 가능하고, 쉽게 작성할 수 있도록 도와주는 몇 가지의 함수 포인터와 정렬 함수들을 작성하는 방법을 배웠습니다. 정렬 함수와 함수 포인터가 효과적인 프로그래밍을 위한 강력하고 융통성 있는 도구이지만, 경우에 따라 모든 매개 변수의 유형을 지정해야 하는 C++의 요구 사항으로 인해 다소 제한적일 때도 있습니다.

예를 들어, 정수를 정렬하는 버블 정렬 프로그램을 이미 개발했다고 가정해 봅시다.

```
void bubblesort(int *list, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++)
            if (list[j + 1] < list[j])
                swap(list[j + 1], list[j]);
    }
}
```

```
}

```

이 함수는 정수를 정렬하는 데에 아주 유용합니다. 그렇다면 이 버블 정렬 함수가 실수(double)에 적용되어야 한다면 어떻게 될까요? 기존 해결책은 이 버블 정렬 함수를 오버로드하여 실수를 위한 버전을 새로 만드는 것입니다:

```
void bubblesort(double *list, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++)
            if (list[j + 1] < list[j])
                swap(list[j + 1], list[j]);
    }
}
```

실수 버전의 버블 정렬 코드는 정수 버전의 버블 정렬 코드와 정확히 동일한 것을 확인할 수 있습니다. 그렇다면 문자를 정렬하고 싶다면 어떻게 될까요? 사용하고 싶은 유형별로 함수를 하나하나 작성해야만 합니다. 모든 유형의 매개변수를 사용할 수 있는 하나의 버블 정렬 함수를 작성할 수 있다면 좋지 않을까요?

템플릿의 세계에 오신 것을 환영합니다^^.

함수 템플릿이란?

C++에서 함수 템플릿은 유사한 다른 함수를 만드는 패턴의 역할을 하는 함수입니다. 함수 템플릿의 기본 아이디어는 일부 또는 모든 변수의 정확한 유형을 지정할 필요 없이 함수를 생성하는 것입니다. 대신, **템플릿 유형 매개 변수**라고 하는 자리 표시자 유형을 사용하여 함수를 정의합니다.

이 자리 표시자 유형을 사용하여 함수를 생성하면, 컴파일러는 하나의 템플릿에서 함수의 여러 “맛”을 만들어낼 수 있습니다!

C++에서 함수 템플릿 생성하기

max() 함수의 정수 버전을 살펴봅시다:

```
int max(int x, int y)
{
    return (x > y) ? x : y;
}
```

특정 유형이 사용되는 세 자리가 있습니다: 매개변수 x , y , 그리고 반환 값은 모두 정수여야 합니다. 함수 템플릿을 생성하기 위해 이 특정 유형을 자리 표시자 유형으로 대체할 것입니다. 이 경우에 교체해야 하는 유형(정수)이 하나뿐이므로 템플릿 유형 매개변수 역시 하나만 필요합니다.

자리 표시자 유형은 예약어가 아닌 한 거의 모든 이름을 원하는 대로 지을 수 있습니다. 하지만, C++에서는 일반적으로 템플릿 유형의 이름을 문자 T(“Type”의 줄임말)로 지정합니다.

다음 함수는 자리 표시자 유형을 이용하여 새로 작성한 함수입니다.

```
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

시작이 좋습니다. 그러나 컴파일러가 “T”가 무엇인지 모르는 상황이므로 컴파일할 수 없습니다!

이 함수가 작동하기 위해서 컴파일러에게 두 가지 사항을 알려줘야 합니다: 첫 번째는 템플릿의 정의이고, 두 번째는 T가 자리 표시자 유형이라는 것입니다. 이 두 가지를 **템플릿 매개변수 선언**을 통해 한 줄로 알려줄 수 있습니다:

```
template <typename T>          // this is the template parameter declaration
T max(T x, T y)
{
    return (x > y) ? x : y;
}
```

믿을지 모르겠지만 이게 필요한 전부입니다. 이제 컴파일할 겁니다!

템플릿 함수가 여러 템플릿 유형 매개변수를 사용하는 경우 심표로 구분할 수 있습니다:

```
template <typename T1, typename T2>
// template function here
```

마지막 참고 사항: 유형 T에 전달된 함수 인수가 클래스 유형일 수 있고, 일반적으로 클래스를 값에 의해 전달(pass by value)하는 것은 좋지 않으므로 템플릿 함수의 매개 변수와 반환 유형을 const 참조(reference)로 만드는 것이 좋습니다:

```
template <typename T>
const T& max(const T& x, const T& y)
{
    return (x > y) ? x : y;
}
```

함수 템플릿 사용하기

함수 템플릿을 사용하는 것은 매우 간단합니다. 여타 함수와 같은 방법으로 사용할 수 있습니다. 다음은 템플릿 함수를 사용한 전체 프로그램입니다:

예시 1:

```
#include <iostream>

template <typename T>
T max(const T x, T y) {
    return (x > y) ? x : y;
}

int main() {
    int i = max(3, 9);           // returns 9
    std::cout << i << '\n';

    double d = max(2.34, 5.67); // returns 5.67
}
```

```
std::cout << d << '\n';

char ch = max('a', 'z');    // returns 'z'
std::cout << ch << '\n';
}
```

예시 2:

```
#include <iostream>

template <typename T>
void bubblesort(T *list, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) // last i elements are already in place
            if (list[j + 1] < list[j])
                swap(list[j + 1], list[j]);
    }
}

template <typename T>
void printlist(T *list, int n) {
    for (int i = 0; i < n; i++) cout << list[i] << " "; cout << endl;
}

int main() {
    // int list[] = { 3, 4, 1, 7, 0, 9, 6, 5, 2, 8};
    char list[] = {'b', 'u', 'b', 'b', 'l', 'e', 's', 'o', 'r', 't'};
    int N = sizeof(list) / sizeof(list[0]);

    cout << "UNSORTED: " << endl;
    printlist(list, N);

    bubblesort(list, N);

    cout << "BUBBLE SORTED: " << endl;
    printlist(list, N);
    cout << "Happy Coding~~\n";
}
```

이진 탐색에 대하여

이진 탐색 알고리즘은 각 재귀 전달별로 배열을 반으로 잘라서 정렬된 배열의 단일 요소를 검색하는 방법입니다. 비결은 배열의 중심에서 중점(midpoint)을 정하고 중점의 값과 찾고 있는 키를 비교한 후 다음 세 가지 조건들 중 하나에 응답하는 것입니다: 키를 중점에서 찾았을 때, 중점의 데이터가 찾고 있는 키보다 클 때, 또는 중점의 데이터가 찾고 있는 키보다 작을 때입니다.

각 전달마다 이전 배열을 반으로 잘라서 새 배열을 생성하므로 이 알고리즘에서는 재귀가 사용됩니다. 이진 탐색 절차를 새로운 (더 작은) 배열에서 재귀적으로 호출합니다. 일반적으로 배열의 크기는 시작 인덱스와 끝 인덱스를 조작하여 조정합니다. 이 알고리즘은 기본적으로 각 전달마다 문제 영역을 반으로 나누므로 로그형 증가 기준(logarithmic order of growth)을 보입니다.

Step 1: binsearch.cpp 구현하기

먼저 제공된 뼈대 코드 binsearch.cpp 를 사용해서 재귀 이진 탐색을 구현하세요.

이 코드에서 사용자는 편의에 따라 `binarysearch(list, size)`와 같이 두 개의 매개변수를 사용해서 함수를 호출할 수 있습니다. `binarysearch(list, size)`는 매 호출마다 새로운 키를 임의로 생성하는 동안 `binarysearch(data, key, lo, hi)`를 "size" 횟수만큼 호출하고 결과를 출력합니다. 키를 리스트에서 찾으면 해당 키의 인덱스를 출력합니다. 키를 찾을 수 없으면 키가 있어야 할 인덱스를 출력합니다. 임의로 정하는 키값의 범위는 리스트의 최솟값과 최댓값 사이입니다.

아시다시피 `binarysearch(list, key, lo, hi)`는 재귀 함수입니다. 모든 재귀 작업은 `binarysearch(list, size)`가 아닌 네 개의 매개 변수가 있는 `binarysearch(list, key, lo, hi)`에서 수행합니다.

다음 코드를 확인하고 제공된 뼈대 코드 `binsearch.cpp` 에서 구현하세요. Step1 을 진행하기 전에 이 기능을 먼저 테스트해보세요.

코드:

```
// This implements a binary search recursive algorithm.
// INPUT: list is an array of integers SORTED in ASCENDING order,
//        key is the integer to search for,
//        lo is the minimum array index,
//        hi is the maximum array index
// OUTPUT: an array index of the key found in the list
//         if not found, return a modified index where it could be found.
int binarysearch(int *list, int key, int lo, int hi) {

    cout << "your code here \n";

    return 0;
}

// randomly generate a key to search between list[0] and list[size-1].
int get_a_key(int *list, int size) {
    int key = rand() % (list[size - 1] + 1 - list[0]) + list[0];
    return key;
}

// calls binarysearch(data, key, lo, hi) "size" number of times
// while generating a new random key at every call of the function.
// and also displays the results. If the key is found in the list,
// it displays its index in the list. If the key is not found, it
// displays where it is supposed to be appeared if there is one.
void binarysearch(int *list, int size) {
    int key = get_a_key(list, size);
    int idx = binarysearch(list, key, 0, size);

    cout << "your code here \n";

    DPRINT(cout << "<binarysearch\n");
}

#if 1
int main(int argc, char *argv[]) {
    int list[] = { 0, 1, 4, 6 };
    // int list[] = { 3, 5, 6, 8, 9, 11 };

    int size = sizeof(list) / sizeof(list[0]);
    srand((unsigned)time(nullptr)); // turn off this line during debugging

    cout << " list: ";
    for (auto x: list)
        cout << x << " ";
    cout << endl;
}
```

```

    for (auto x: list)
        binarysearch(list, size);
}
#endif

```

main() 함수가 두 개의 **binarysearch()** 함수를 테스트하기 위해 제공됩니다.

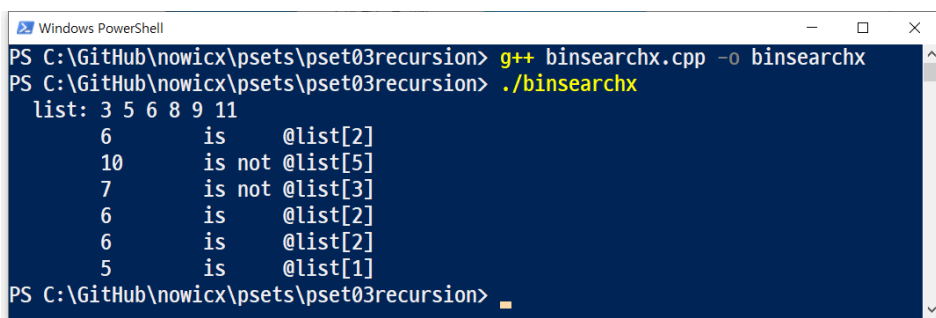
- **rand()**, **srand()** 그리고 a 와 b 사이의 난수를 사용하는 법은 한국어로 된 [이 웹사이트](#)를 참고하세요 (해당 웹사이트의 포스트를 더 이상 확인할 수 없습니다).

알고리즘 테스트하기

binarysearch()를 테스트하기 위한 임의의 숫자 또는 키를 생성하려고 합니다.

- 키값은 리스트의 첫 번째 값과 마지막 값 사이 또는, 이 경우에는, [3 ... 11] 사이입니다.
- 리스트에 있는 원소의 수만큼 반복하여 테스트합니다.
- 결과를 아래와 같이 출력합니다.

실행 예시:



```

Windows PowerShell
PS C:\GitHub\nowicx\psets\pset03recursion> g++ binsearchx.cpp -o binsearchx
PS C:\GitHub\nowicx\psets\pset03recursion> ./binsearchx
list: 3 5 6 8 9 11
6      is    @list[2]
10     is not @list[5]
7      is not @list[3]
6      is    @list[2]
6      is    @list[2]
5      is    @list[1]
PS C:\GitHub\nowicx\psets\pset03recursion>

```

- 첫 번째 줄은 6 을 list[2]에서 찾았음을 나타냅니다. 두 번째 줄은 10 을 찾을 수 없음을 나타내고, 만약 찾았다면 list[5]에서 찾았을 것이라는 걸 나타냅니다. 다른 줄도 동일한 설명이 적용됩니다.

Take-away: n 개의 항목을 **binarysearch(list, key, lo, hi)**하려면 시간이 얼마나 소요될까요?

binarysearch()를 한 번 호출하면 최소 절반의 요소를 고려 대상에서 제거합니다. 따라서 가능성을 1 로 줄이기 위해서는 **binarysearch()** 호출이 $\log_2 n$ (밑이 2 인 로그 n)번 사용됩니다. 그러므로 **binarysearch()**는 $\log_2 n$ 에 비례하는 시간이 소요됩니다.

Step 2: binsearchT.cpp - 템플릿 버전

이제 정수뿐만 아니라 문자 데이터로도 작동하는 **binsearch.cpp** 의 템플릿 버전을 구현하려고 합니다. 이 새 버전 **binsearchT.cpp** 는 **main()**의 **리스트** 데이터 유형을 제외한 이진 탐색 함수를 일체 수정하지 않고 문자형 **리스트** 및 정수형 **리스트**와 작동해야 합니다.

아래의 **main()** 함수는 입력 리스트 데이터의 유형을 문자형 또는 정수형으로 바꾸는 것을 제외하고는 어떤 것도 변경하지 않습니다. 사용자가 **char list[]** 또는 **int list[]** 중 어떤 유형을 사용하더라도 이 파일의 이진 탐색 함수를 수정하지 않은 상태로 아래에 있는 **main()** 함수와 작동해야 합니다.

```

#if 1
int main(int argc, char *argv[]) {
    // char list[] = { 'b', 'e', 's', 't' };
    char list[] = { 'a', 'c', 'e', 'g', 'i', 'k' };

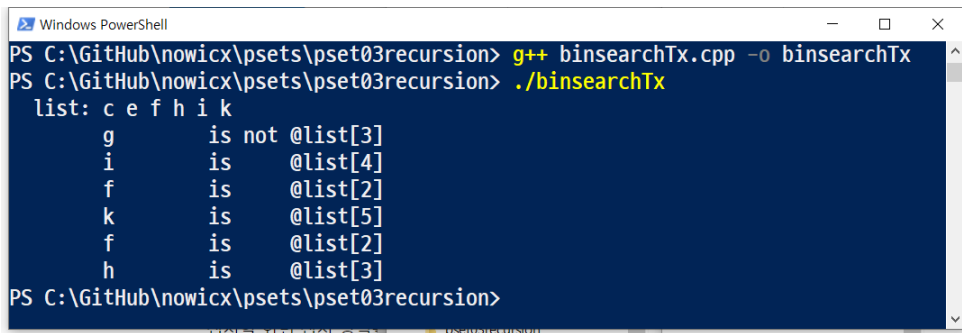
```

```
// int list[] = { 0, 1, 4, 6 };
// int list[] = { 3, 5, 6, 8, 9, 11 };

int size = sizeof(list) / sizeof(list[0]);
srand((unsigned)time(nullptr)); // turn off this line during debugging

cout << " list: ";
for (auto x: list) cout << x << " ";    cout << endl;
for (auto x: list)
    binarysearch(list, size);
}
#endif
```

실행 예시:



```
Windows PowerShell
PS C:\GitHub\nowicx\psets\pset03recursion> g++ binsearchTx.cpp -o binsearchTx
PS C:\GitHub\nowicx\psets\pset03recursion> ./binsearchTx
list: c e f h i k
g      is not @list[3]
i      is      @list[4]
f      is      @list[2]
k      is      @list[5]
f      is      @list[2]
h      is      @list[3]
PS C:\GitHub\nowicx\psets\pset03recursion>
```

- 첫 번째 줄은 'g'를 찾을 수 없음을 나타냅니다. 만약 찾았다면 list[3]에서 찾았을 것이라는 것을 나타냅니다. 두 번째 줄은 'i'를 list[4]에서 찾았음을 나타냅니다. 다른 줄도 동일한 설명이 적용됩니다.

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. 다시 제출하는 횟수에는 제한이 없습니다. 마감 기한 전에 가장 마지막으로 제출된 파일을 채점할 것입니다.

제출 파일 목록 & 배점

다음 파일들을 piazza 폴더에 제출하세요

- binsearch.cpp
- binsearchT.cpp

마감 기한

- 마감 기한: 11:55 pm