자료구조실습 보조 강의자료 12

본격적으로 C++ 코딩을 진행하기에 앞서 몇 가지 컨테이너를 살펴보고 넘어갑시다.

**priority_queue**

힙 자료구조를 구현하는 컨테이너 어댑터입니다.

`#include <queue>`

| **Member functions** | |
|---|---|
| (constructor) | constructs the `priority_queue` |
| (destructor) | destructs the `priority_queue` |
| operator= | assigns values to the container adaptor |
| **Element access** | |
| top | accesses the top element |
| **Capacity** | |
| empty | checks whether the underlying container is empty |
| size | returns the number of elements |
| **Modifiers** | |
| push | inserts element and sorts the underlying container |
| emplace | constructs element in-place and sorts the underlying container |
| pop | removes the top element |
| swap | swaps the contents |

**Practice**

```
const auto data = { 1, 3, 3, 5, 2, 9, 7, 1 };
for (auto& i : data)
        cout << i << " ";
cout << endl;

priority_queue<int> Q;
for (auto& i : data)
        Q.push(i);
for (cout << "priority queue: "; !Q.empty(); Q.pop())
        cout << Q.top() << " ";
cout << endl;

priority_queue<int, vector<int>, greater<int>>
        MinQ(data.begin(), data.end());
for (cout << "priority queue: "; !MinQ.empty(); MinQ.pop())
        cout << MinQ.top() << " ";
cout << endl;
```

priority_queue는 단순하고, 쓰임이 많습니다. 특히 간단히 자료구조를 구성하여 순차 정렬하거나, 가장 큰 값(또는 작은 값)을 하나씩 꺼내 올 때 매우 유용합니다.

**확인하고 넘어갈 내용들**

- push, top, pop 등의 쓰임
- 최대 힙, 최소 힙 구성 방법

**set**

유니크 한 키(Key) 값을 갖는 데이터를 다루는 연관 컨테이너입니다.

#include <set>

| **Member functions** | |
|---|---|
| (constructor) | constructs the `set` |
| (destructor) | destructs the `set` |
| operator= | assigns values to the container |
| get_allocator | returns the associated allocator |
| **Iterators** | |
| begincbegin | returns an iterator to the beginning |
| endcend | returns an iterator to the end |
| rbegincrbegin | returns a reverse iterator to the beginning |
| rendcrend | returns a reverse iterator to the end |
| **Capacity** | |
| empty | checks whether the container is empty |
| size | returns the number of elements |
| max_size | returns the maximum possible number of elements |
| **Modifiers** | |
| clear | clears the contents |
| insert | inserts elements or nodes (since C++17) |
| emplace | constructs element in-place |
| emplace_hint | constructs elements in-place using a hint |
| erase | erases elements |
| swap | swaps the contents |
| extract | extracts nodes from the container |
| merge | splices nodes from another container |
| **Lookup** | |
| count | returns the number of elements matching specific key |
| find | finds element with specific key |
| contains | checks if the container contains element with specific key |
| equal_range | returns range of elements matching a specific key |
| lower_bound | returns an iterator to the first element *not less* than the given key |
| upper_bound | returns an iterator to the first element *greater* than the given key |
| **Observers** | |
| key_comp | returns the function that compares keys |
| value_comp | returns the function that compares keys in objects of type value type |

**Practice**

```
set<unsigned> dsp;
cout << "empty: " << dsp.empty() << endl;
```

```
dsp.insert(1);
cout << "empty: " << dsp.empty() << endl;
cout << "size: " << dsp.size() << endl;
cout << "max_size: " << dsp.max_size() << endl;
dsp.insert(2);
dsp.insert(2);
dsp.insert(3);
for (auto& i : dsp)
        cout << i << " ";
cout << endl;
cout << "count key value 2: " << dsp.count(2) << endl;
cout << "contains key value 2: " << dsp.contains(2) << endl;
dsp.insert(-3);
for (auto& i : dsp)
        cout << i << " ";
cout << endl;

dsp.clear();
for (auto& i : dsp)
        cout << i << " ";
cout << endl;
cout << "empty: " << dsp.empty() << endl;
cout << "size: " << dsp.size() << endl;
```

set은 특별한 기능이 많지 않지만, 유니크 한 키 값을 다루기 때문에 단순한 자료 구조 표현에 자주 사용합니다. 별 것 없는데 자주 보이는 컨테이너입니다. 단순한 만큼 사용이 쉬우니 알아 두세요. 단순한 컨테이너이기 때문에 unsigned 자료형도 함께 예시로 넣었습니다.

**확인하고 넘어갈 내용들**

- insert, find, contains 등의 쓰임
- unsigned 자료형의 표현 방법

**map**

유니크 한 키 값(key)과 대응하는 값(value)으로 페어(pair)를 만들어 구성하는 연관 컨테이너입니다.

`#include <map>`

| | |
|---|---|
| **Member functions** | |
| (constructor) | constructs the `map` |
| (destructor) | destructs the `map` |
| operator= | assigns values to the container |
| get_allocator | returns the associated allocator |
| **Element access** | |
| at | access specified element with bounds checking |
| operator[] | access or insert specified element |
| **Iterators** | |
| begincbegin | returns an iterator to the beginning |
| endcend | returns an iterator to the end |
| rbegincrbegin | returns a reverse iterator to the beginning |
| rendcrend | returns a reverse iterator to the end |
| **Capacity** | |
| empty | checks whether the container is empty |
| size | returns the number of elements |
| max_size | returns the maximum possible number of elements |
| **Modifiers** | |
| clear | clears the contents |
| insert | inserts elements or nodes (since C++17) |
| insert_or_assign | inserts an element or assigns to the current element if the key already exists |
| emplace | constructs element in-place |
| emplace_hint | constructs elements in-place using a hint |
| try_emplace | inserts in-place if the key does not exist, does nothing if the key exists |
| erase | erases elements |
| swap | swaps the contents |
| extract | extracts nodes from the container |
| merge | splices nodes from another container |
| **Lookup** | |
| count | returns the number of elements matching specific key |
| find | finds element with specific key |
| contains | checks if the container contains element with specific key |
| equal_range | returns range of elements matching a specific key |
| lower_bound | returns an iterator to the first element *not less* than the given key |
| upper_bound | returns an iterator to the first element *greater* than the given key |

**Practice**

```
map<string, int> m{ {"Brazil", 333}, {"Argentina", 550}, {"France", 750} };
m["England"] = 820;
for (auto& [key, value] : m)
        cout << "key: " << key << ", value: " << value << endl;
m["England"] = 800;
cout << m["Spain"] << endl;
for (auto& [key, value] : m)
        cout << "key: " << key << ", value: " << value << endl;
erase_if(m, [](const auto& pair) {
        return pair.second == 0;
        });
for (auto& [key, value] : m)
        cout << "key: " << key << ", value: " << value << endl;
```

set은 특별한 기능이 많지 않지만, 유니크 한 키 값을 다루기 때문에 단순한 자료 구조 표현에 자주 사용합니다. 별 것 없는데 자주 보이는 컨테이너입니다. 단순한 만큼 사용이 쉬우니 알아 두세요. 단순한 컨테이너이기 때문에 unsigned 자료형도 함께 예시로 넣었습니다.

**확인하고 넘어갈 내용들**

- insert, find, contains 등의 쓰임
- unsigned 자료형의 표현 방법

**보너스 하나: numeric_limits**

전역변수로 INF 값을 애매하게 설정하고 코드를 작성하는 과정이 사실 아쉬울 수 있습니다. 알고 있는 변수 형태의 최대값과 다르니까요. 그리고 그래프를 만든 뒤 변수 형태가 달라질 때마다 다르게 설정해 줘야 하나? 라는 생각도 들 수 있습니다. 이를 해결하기 위한 키워드로 numeric_limits가 있습니다.
사용방법부터 확인해 봅시다.

Practice

```
numeric_limits<int>::lowest();    // -2147483648
numeric_limits<int>::min();       // -2147483648
numeric_limits<int>::max();       // 2147483647
numeric_limits<float>::lowest();  // -3.40282e+38
numeric_limits<float>::min();     // 1.17549e-38
numeric_limits<float>::max();     // 3.40282e+38
numeric_limits<double>::lowest(); // -1.79769e+308
numeric_limits<double>::min();    // 2.22507e-308
numeric_limits<double>::max();    // 1.79769e+308
```

이제 변수 타입만 template으로 바꾸면, 코드를 중복해 사용할 필요 없이 정확하게 그래프를 표현할 수 있습니다. template으로 바꾼 코드는 수업 시간 코드를 확인해 보세요.

**보너스 둘: 참조 반환**

참조가 익숙하지 않은 경우를 위해 준비했습니다. 함수 반환형을 참조할 때, 반환하는 변수가 참조가 가능하고, 참조해야 하는 경우는 참조해야 합니다. 참조는 서로 다른 영역(서로 다른 함수)에 있는 변수를 연결시켜 주는 역할을 합니다. 두 예시를 통해 확인해 봅시다.

Practice 1

```
int& func(int x){
        int value = x + 5;
        return value;
}
int main() {
        int a = 0;
        cout << func(a);
        return 0;
}
```

코드는 작동은 합니다. 하지만 func 함수에서 사용한 변수 value는 func 호출이 끝나면 사라져야 하지만, main 함수에서 이 변수를 참조하고 있습니다. 최근 C++ 버전에서는 이런 경우에도 값을 유지를 시켜주는 안전 장치가 있어, 경고 메시지만 띄우고 코드 작동은 가능합니다. 하지만 불안정하죠?

Practice 2

```
int func(array<int, 10>& array, int index) {
    return array[index];
}
int main() {
    array<int, 10> array;
    func(array, 3) = 2;           // 이 곳
    cout << array[3] << endl;
    return 0;
}
```

func 함수는 array 객체를 매개변수로 참조해서 가져오고, 이를 다시 반환합니다. 그러나 참조해서 가져온 객체이기 때문에 func 함수가 이를 참조하지 않고 반환할 경우, main 함수에서는 이를 수정할 수 없는 값이 됩니다. 따라서 코드가 작동하지 않습니다.

'// 이 곳' 라인을 다시 설명하면, func(array, 3)은 array의 3번째 요소에 접근하여, 이 위치에 값 2를 넣습니다. func 함수는 main 함수의 array를 참조해 가져와서, 특정 위치(index) element를 반환하지만, func과 다른 함수인 main 함수에서는 해당 위치를 연결해 주지 못합니다. 그런데 func 함수에서 이를 참조하지 않고 반환했기 때문에, main 함수에서는 func 이 반환한 element에 직접 연결할 수가 없고, 값도 넣을 수가 없습니다. 따라서 lvalue 오류 메시지를 출력합니다.

**보너스 셋: emplace**

emplace는 생성과 동시에 삽입을 합니다. 즉 생성자를 활용합니다. 흔히 비슷하게 쓰이는 insert, push_back은 object를 copy and move하고, emplace, emplace_back은 생성자를 활용합니다.

덜 익숙한 emplace의 활용만 보고 넘어갑니다(아래 예시는 emplace와 insert 모두 동일하게 작동합니다).

Practice

```
vector<int> myvector = { 10,20,30 };
auto it = myvector.emplace(myvector.begin() + 1, 100);
myvector.emplace(it, 200);
myvector.emplace(myvector.end(), 300);
cout << "myvector contains:";
for (auto& x : myvector)
    cout << ' ' << x;
```

벡터는 10, 20, 30으로 초기화했습니다. 이터레이터는 두번째 위치로 설정합니다. 그리고 100을 넣습니다. 벡터는 10, 100, 20, 30이 됩니다. 그리고 같은 이터레이터에 200을 넣습니다. 이터레이터를 변경하지 않았기에, 여전히 같은 위치(두번째 위치)에 200을 넣습니다. 벡터는 10, 200, 100, 20, 30이 됩니다.

이번에는 끝에 300을 넣습니다. 벡터는 10, 200, 100, 20, 30, 300이 됩니다. emplace를 insert로 변경해도 결과는 같습니다.

**보너스 넷: istringstream**

string을 parsing(구성을 분해하여 새로 구조를 결정)합니다. 사용법은 하나 정도만 보면 알 수 있습니다.

```
istringstream iss("data structure, 21 Nov.");
string s1, s2, s3;
int date;
iss >> s1 >> s2 >> date >> s3;
cout << s1 << ' ' << s2 << ' ' << date << ' ' << s3 << endl;
```

이렇게 문자열을 띄어쓰기로 나눠서 구성할 수 있습니다.