STL 기초

Modern C++

한컴에듀케이션

목 차

- pair
- reference
- auto
- template
- range based for loop
- initializer list
- class *vs* struct
- mem function
- operator overloading
- function object

pair

- 두개의 값을 하나의 쌍으로 관리해주는 하나의 data type
- #include<utility.h> (container 사용시 생략 가능)
- pair<T, U> p pair<int, double> p = { 2, 3.1 };
- p.first, p.second 로 인자 접근 : p.first = 2 , p.second = 3.1
- operator (==, !=, <, >, <=, >=) 정의되어 있음
  == : first, second 가 둘 다 같다
  < : 1순위. first가 작다 , 2순위. second가 작다

reference - 참조자

- 자신이 참조하는 변수를 대신할 수 있는 또 하나의 이름
- 생성시 초기화 필요
- reference 참조 , 8byte copy, 원본 변경 가능

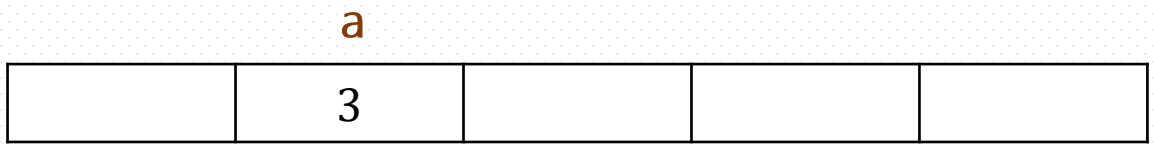
```
int a;
int&c = a;
```

```
int arr[10];
for (int&x : arr) {...}
```

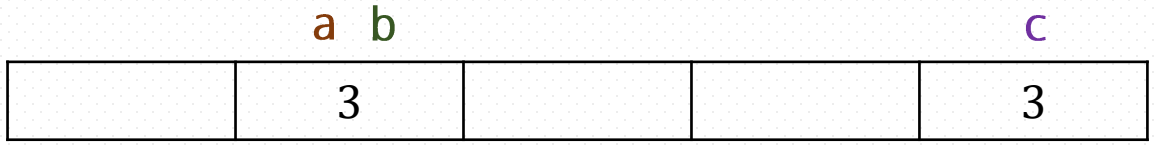
```
void func(int&b) {...}

int a;
func(a);
```

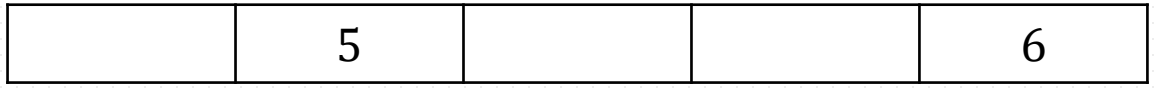
```
int a = 3;
```




```
int &b = a;
int c = a;
```




```
b = 5;
c = 6;
```



auto

- 초기값 type에 맞춰 선언하는 변수의 type이 자동으로 결정
- compile time에 자동으로 type 완성 (선언과 동시에 초기화 필요)
- 함수의 return type 지원
- 함수의 매개변수에는 지원 안됨 (c++14 기준)　　: 필요시, template 사용
- iterator 등 type이 복잡해지는 경우에 특히 유용하게 사용

```
auto a = 1;          : int
auto &b = a;         : int
```

Template

- 여러 자료형을 하나의 함수, 클래스로 사용 할 수 있게 만들어 놓은 틀
- typename , class keyword는 완전 동일한 기능
- stl(standard **template** library)
- 형식 매개 변수 여러 개 사용 가능

```
template <typename T, typename U, typename V> class Template{};
```

Function template

```
template <typename T>
T sum(T a, T b) { return a + b; }

        sum(1, 2);
        sum(1.3, 2.8);
        sum('a', 'b');
```

Class template

```
template<class T>
struct Data { T a, b; };

        Data<int> A;
        Data<double> A;
        Data<char> A;
```

range-based for loop

for(① : ②) { … } : 범위 기반 for loop

②의 시작부터 끝까지 각 원소를 ①에 담으면서 loop 수행

① : 배열, 컨테이너의 각 원소를 담을 변수 선언

② : 구간을 반복할 배열, 컨테이너 이름

```
int arr[5] = { 1, 2, 3, 4, 5}

for(auto x : arr) { x++; }
```

- copy value : 객체크기만큼 복사, 원본 변경 불가
- 수행 후 arr : 1, 2, 3, 4 ,5

```
int arr[5] = { 1, 2, 3, 4, 5}

for(auto& x : arr) { x++; }
```

- copy reference : 8byte 복사, 원본 변경 가능
- 수행 후 arr : 2, 3, 4, 5 ,6

class *vs* struct

- 특정 객체를 생성하기 위해 멤버 변수와 멤버 함수를 정의하는 일종의 틀
- C++ 에서는 class와 struct의 기능은 한가지를 제외하고 완전히 동일
- 접근지시자 default 값 = (Class : private , struct : public)
- uniform initialization 적극 활용

```
struct Data {
    int x, y;
    void print() {
        printf("%d %d\n", x, y);
    }
    int sum() {
        return x + y;
    }
    int mul() {
        return x * y;
    }
};
```

```
Data A = {};     // x=0, y=0
Data B = {1};    // x=1, y=0
Data C = {2,3}; // x=2, y=3
```

Initializer list 초기화자 리스트

- braced-init-list {…} 로 원소들을 담은 type
- braced-init-list 가 초기화 또는 대입에 사용되는 경우, auto에 바인딩 되는 경우 자동 생성
- Narrow-conversion(암시적 타입 변환, 데이터 손실이 있는 변환) 불가
- 모든 class의 기본 생성자 및 대입연산의 인자로 initializer_list type이 제공됨
  따라서, STL container를 포함한 모든 객체를 {} 활용해 생성 및 대입 가능

```
initializer_list<int> il{ 1,2,3,4,5 };
for (auto x : il) cou << x << ' ';
```

```
set<int> s{ 1,2,3,4,5 };                    // 1,2,3,4,5
s = { 5, 6 };                                // 5,6
vector<pair<int, int>> v{ {1,2},{3,1},{2,5} };
```

mem fucntion

- #include<string.h>
- byte 단위로 원하는 기능 수행
- loop 보다 빠름

```
void* memset( void* dest, int ch, std::size_t count );
```

dest의 count개의 byte 값을 ch로 변경
보통 0 초기화시에 자주 사용
ex) memset(arr, 0, sizeof(arr))

```
int memcmp( const void* lhs, const void* rhs, std::size_t count );
```

lhs와 rhs의 count byte 크기의 값이 같은지 비교
ex) memcmp(arr, arr2, sizeof(arr))

```
void* memcpy( void* dest, const void* src, std::size_t count );
```

src의 count byte 크기를 src로 복사
ex) memcpy(arr, arr2, sizeof(arr))

Operator

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	++a --a a++ a--	+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	!a a && b a || b	a == b a != b a < b a > b a <= b a >= b a <=> b	a[b] *a &a a->b a.b a->*b a.*b	a(...) a, b ? :

Operator Overloading

- primitive type은 연산자를 default로 사용 가능하다.

```
int a=3, b=4;

a == b   : true
a + b    : 7
a > b    : false
```

- custom data type은??

```
struct Data { int x, y; };

Data a, b;

a == b ?
a + b ?
a > b ?
```

특정한 기준이 없으므로 연산자를 사용할 수 없다.

이를 정의해주는게 operator overloading(연산자 오버로딩)

Operator Overloading

Member function Overloading (global function overloading은 생략)

```
struct Data {
    int x, y;
    bool operator==(Data r) {
        return x == r.x && y == r.y;
    }
    bool operator<(Data r) {
        if (x != r.x) return x < r.x;
        return y < r.y;
    }
    void operator()() {
        cout << x << ' ' << y;
    }
    void operator()(Data r) {
        x += r.x;
        y += r.y;
    }
};
```

```
Data a, b;


a == b   : a.operator==(b)
a < b    : a.operator<(b)
a()      : a.operator()()
a(b)     : a.operator()(b)
```

※ operator() : function call operator

함수를 인자로

- ~~function pointer~~
- function object
- ~~lambda~~

STL container에서 기준 설정할 때 활용(다른 방법도 있지만 우리에게는 이거 밖에 없다고 가정)

- `set<T, Compare>`
- `unordered_set<T, Hash, KeyEqual>`

※ `Compare`, `Hash`, `KeyEqual` : function object

Function object (functor)

- function call operator()를 정의하여 함수처럼 사용가능한 객체
- 인라인 치환
- 상태를 가질 수 있다
- 다른 함수의 인자로 전달될 수 있다.

```
struct Sum_n {
    int operator()(int n) {
        int sum = 0;
        for (int i = 1; i <= n; i++) sum += i;
        return sum;
    }
}sum_n;
```

```
cout << sum_n(10);        // 55
cout << Sum_n{}(10);      // 55
cout << Sum_n()(10);      // 55
```

Predefined Functor : comparision operator

- set, map, priority_queue 에서 default로 less 사용
- unordered_set, unordered_map에서 default로 equal_to 사용
- data type에 각각 operator==, operator<, operator>가 정의되어 있어야 한다.

1. equal_to

```
template<class T>
struct equal_to {
  bool operator()(const T& lhs, const T& rhs) const {
    return lhs == rhs;
  }
}
```

2. less

```
template<class T>
struct less {
  bool operator()(const T& lhs, const T& rhs) const {
    return lhs < rhs;
  }
}
```

3. greater

```
template<class T>
struct greater {
  bool operator()(const T& lhs, const T& rhs) const {
    return lhs > rhs;
  }
}
```

형태

```
1. less<T>          : class

2. less<T>()        : function

    less<int>()(1,1) = false
    less<int>()(1,2) = true

3. less<T>{}        : function

    less<int>{}(1,1) = false
    less<int>{}(1,2) = true
```

Predefined Functor : hash

- **Requirement**
  - key값에 대해 size_t type의 hash 값을 반환한다.

    (size_t == unsigned long long)
  - k1, k2가 같다면 hash<Key>(k1) == hash<Key>(k2) 이어야 한다.
  - k1, k2가 다르다면 hash<Key>(k1) == hash<Key>(k2) 인 확률이 매우 적어야 한다.

- **specialization 되어 있는 type**
  - 기본 primitive type (int, long long, char, bool, double, float, …)
  - string
  - ~~pair~~

※ specialization : 특정 data type에 대해 별도의 동작을 정의해준다.
즉, 각 type 특성에 맞게 hash function 이 적절히 구현되어 있다.

※ Key값을 받아, hash value를 반환한다.

```
struct hash {
    size_t operator()(const Key &key) const{
        return (hash value);
    }
};
```

lambda

익명 함수 | 객체 함수

```
[captures](parameters) -> return type { body };
```

①　　②　　③　　④

① : scope 내의 외부 변수를 reference or value로 참조 가능

② : 매개 변수 (없으면 생략 가능)

③ : 생략시 body의 return type으로 결정

④ : 함수 본문

captures 활용

[]　　: 외부 변수 사용 X
[&]　　: scope 내의 모든 외부 변수들을 레퍼런스로 가져온다.
[=]　　: scope 내의 모든 외부 변수들을 값으로 가져온다.
[=, &x]　　: x (reference 참조) | 나머지 (value 참조)
[x, &y, &z] : x (value 참조)　　| y, z (reference 참조)

lambda

```
auto sum_n = [](int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) sum += i;
    return sum;
};

sum_n(10); // result : 55
```

```
int arr[5]{ 1,2,3,4,5 };

auto increase = [&] {
    for (int i = 0; i < 5; i++) arr[i]++;
};

increase(); // arr : 2,3,4,5,6
```


감사합니다