12주차 표준 템플릿, STL

템플릿(template)

- 매개변수타입만 다른 중복된 함수들을 일반화시킨 틀(tamplate)
- 함수나 클래스를 일반화하는 도구
- **함수의 작성을 용이**하게 하고 **코드의 재사용**을 가능하게하므로 **소프트웨어 생산성**과 **유연성**을 높이는 효과, 하지만 **포팅에 취약**, 오류메시지가 빈약해 **디버깅에 어려움**이 있을 수 있다.

템플릿 정의

- `template <typename T>`와 같은 형태로 템플릿을 정의
- `typename` 대신에 `class` 키워드를 사용할 수도 있다. `template <class T>`

```
ex)10-1
template <class T> // === <typename T>
void myswap(T &a, T &b){
T temp = a;
a = b;
b = temp;
}
```

코드 구체화

- 중복함수들을 템플릿화 하는 과정의 역과정을 **구체화** 라고 부름
- 컴파일러가 함수의 호출문을 컴파일 할 때, 구체화를 통해 소스코드를 생성해 냄
- 이렇게 생성된 함수를 **구체화된 함수**라고 부름

코드 구체화시 주의점

- 함수를 구체화시키는 과정에서 제네릭 타입 T에 유의해야함
- 매개변수 a,b 모두 타입 T로 선언되어있기 때문, 두개의 타입이 동일해야함!
- 타입이 다르면 컴파일 오류.

```
int s=1; double t=5;
myswap(s,t); // compile error
```

제너릭 함수 생성

- 클래스 선언부와 구현부를 모두 template로 선언
- 제네릭 클래스의 멤버 함수는 자동 제네릭 함수임.
  / 큰 값을 비교하는 제너릭 함수 bigger

```
ex)10-2
template <class T>
T bigger(T a, T b) // T는 타입을 의미한다.
// 여기선 반환 타입이 같으므로 T bigger로 표현했지만,
// 반환 타입이 다르면 다른 타입으로 표현해야 한다.
```

```
{
    if (a > b)
        return a;
    else
        return b;
}
```

/ 배열의 n 번째까지 더하는 제너릭 함수 add

```
template <class T>
T add(T data[], int n){ // (첫 타입은 T, 두번째는 항상 int)
    T sum = 0; // int 면 int sum, double 이면 double sum 으로 변환.
    for (int i = 0; i < n; i++)
        sum += data[i];
    return sum;
}
int main(){
    int x[] = {1, 2, 3, 4, 5};
    double y [] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};

    cout << "배열 x의 합 = " << add(x, 5) << endl;
    cout << "배열 y의 합 = " << add(y, 7) << endl;
}
```

- 제네릭 함수와 중복함수가 **동시에 선언**되었을경우 **중복함수**가 **템플릿함수보다 우선** 바인딩
- 템플릿 함수에도 디폴트 매개변수를 정의해 둘 수 있다

제너릭 클래스

- 제너릭함수와 마찬가지로 제너릭 클래스도 가능

```
template <class T>
class MyStack{
        int tos;
        T data[100];
public: // 선언부
        MyStack();
        void push(T element);
        T pop();
}
```

두개의 제네릭 타입을 가진 클래스 생성. (중요)

```
#include <iostream>
using namespace std;

template <class T1, class T2> // 두 개의 제네릭 타입 선언
class GClass
{
```

```
    T1 data1;
    T2 data2;

public:
    GClass();
    void set(T1 a, T2 b);
    void get(T1 &a, T2 &b);
};

template <class T1, class T2>
GClass<T1, T2>::GClass()
{
    data1 = 0;
    data2 = 0;
}

template <class T1, class T2>
void GClass<T1, T2>::set(T1 a, T2 b)
{
    data1 = a;
    data2 = b;
}

template <class T1, class T2>
void GClass<T1, T2>::get(T1 &a, T2 &b)
{
    a = data1;
    b = data2;
}

int main()
{
    int a;
    double b;

    GClass<int, double> x;

    x.set(2, 0.5);
    x.get(a, b);
    cout << "a=" << a << '\t' << "b=" << b << endl;

    char c;
    float d;

    GClass<char, float> y;

    y.set('m', 12.5);
    y.get(c, d);
    cout << "c=" << c << '\t' << "d=" << d << endl;
}
```

STL 라이브러리

Vector

- 가변 길이 배열을 구현한 **제네릭 클래스**, 스스로 내부 크기를 조절하므로 크기 고민 x
- 인덱스는 0부터 시작하며 **제네릭 클래스**이기때문에 **타입을 지정**해서 선언해야함

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> v; // int 형 벡터 선언

    v.push_back(1); // push_back(n) n 요소 삽입
    v.push_back(2);
    v.push_back(3);

    for(int i=0; i<v.size(); i++){ // size() 벡터의 원소 개수
        cout<<v[i]<<endl; // 배열처럼 접근가능
    }
    v[0] = 10;
    int m = v[2];
    v.at(2)=5; // at(n) n번째 인덱스에 접근

    for(int i=0; i<v.size(); i++){
        cout<<v[i]<<endl;
    }
}
```

iterator

- **반복자** 라고도 부르며, 컨테이너의 원소를 가르키는 **포인터**
- **구체적인 컨테이너를 지정**해 반복자 변수 생성

```
vector<int>::iterator it; // it 자체가 포인터
it = v.begin(); // vertor 배열의 첫번째 원소에 접근 begin()
```

```
int main(){
    vector<int> v; // int형 벡터 선언
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    vector<int>::iterator it; // 반복자 선언
        // vector<int>::iterator it = v.begin(); 와 같음

    for(it = v.begin(); it != v.end(); it++){
    // 모든 원소에 2를 곱하는 코드
        int n = *it; // it가 가리키는 값!!! 을 n에 저장
        n = n*2; // 2*2
        *it = n; // 다시 it가 가리키는 값!!! 에 n을 저장
    }

    for(it = v.begin(); it != v.end(); it++){
        cout << *it << endl; // 전체 출력
```

```
    }
} // 결과 : 2 4 6
```

... 508 P
영한사전 예제 10-12)

map

- ('키','값')의 쌍을 원소로 저장하는 제네릭 컨테이너
- 동일한 '키'를 가진 원소가 중복 저장되면 오류 발생
- '키'로 '값' 검색
- 많은 응용에서 필요

논의한점 : 어떤상황에서 템플릿과 제네릭 프로그래밍을 사용하면 좋을까?
-> 동일한 로직을 가지지만 다양한 데이터 타입에 대해 동작해야 하는 함수를 작성할 때 템플릿을 사용하는 것이 효과적일것

-> 템플릿을 사용할 때 주의해야 할 사항은 템플릿은 코드의 일반화를 가능하게 하지만, 함수를 구체화할 때에 제네릭 타입에 유의해야한다는 점임. 모든 매개변수가 동일한 타입이여야 한다는것이 중요함.

-> 알고리즘을 일반화하는 코드를 작성할때 매우 유용하게 사용가능함, 다양한 데이터 타입에 동일한 구조로 일반화 할 수 있음.

-> 하지만 템플릿을 너무 남용하는경우 오히려 코드 가독성이 감소할것같음, 제네릭 타입을 명확하게 작성해주는 절차가 필요해 보임.