# 13. Template 1

13주차

526-533p

# 템플릿에 대한 이해와 함수 템플릿

- 함수 template
  - 컴파일러가 필요시에 타입 확정하여 함수를 생성하도록 정의한 것
- Template 함수
  - 함수 template을 보고 컴파일러가 생성한 함수
  - 처음 필요한 시점에 1번만 생성함
- Template함수는 일반함수와 구분됨
  - 같은 형태의 함수가 Template함수와 일반함수로 각각 정의되어도 됨.
  - 이렇게 사용하지 말자.

526-533p

# 템플릿에 대한 이해와 함수 템플릿

```
#include <iostream>
using namespace std;
template <typename T>
T Add(T num1, T num2) {
        return num1 + num2;
}
int main(void) {
        cout<< Add<int>(15, 20) <<endl;
        cout<< Add<double>(2.9, 3.7) <<endl;
        cout<< Add<int>(3.2, 3.2) <<endl;
        cout<< Add<double>(3.14, 2.75)  <<endl;
        return 0;
}
int main(void) {
        cout << Add(15, 20) << endl;
        cout << Add(2.9, 3.7) << endl;
        cout << Add(3.2, 3.2) << endl;
        cout << Add(3.14, 2.75) << endl;
        return 0;
}
```

함수 template.

template <class T>
형태도 가능!

원래는 타입을 지정하여 호출해야 하지만...

형을 생략하면 컴파일러가 판단하여 적절한 타입의 template 함수를 생성해 준다.

533-535p

# 둘 이상의 타입에 대해서 템플릿 선언하기

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

template <class T1, class T2>
void ShowData(double num)
{
        cout << (T1)num << ", " << (T2)num << endl;
}

int main(void)
{
        ShowData<char, int>(65);
        ShowData<char, int>(67);
        ShowData<char, double>(68.9);
        ShowData<short, double>(69.2);
        ShowData<short, double>(70.4);
        return 0;
}
```

둘 이상의 타입을 , 로 연결하여 사용 가능.

C++에서는 아래와 같이 형변환도 가능.
cout << T1(num) << ", " << T2(num) << endl;

매개변수 형이 double이어서 매개변수로 타입을 유추할 수 없으므로 호출 형식을 완전히 갖춰서만 호출 가능.

# 함수 템플릿의 특수화
## (SpecialFunctionTemplate.cpp 1/2)

536-539p

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

template <typename T>
T Max(T a, T b) {
    cout << "T Max(T a, T b)" << endl;
    return a > b ? a : b;
}

template <>
char* Max<char *>(char* a, char* b) {
    cout << "char* Max<char*>(char* a, char* b)" << endl;
    return strlen(a) > strlen(b) ? a : b;
}

template <>
const char* Max<const char *>(const char* a, const char* b) {
    cout << "const char* Max<const char*>(const char* a, const char* b)" << endl;
    return strcmp(a, b) > 0 ? a : b;
}
```

두 데이터를 비교하여 더 큰 쪽을 반환하는 함수 template.
데이터가 문자열일 때의 별도 처리가 필요!

함수 template 에서 타입이 char * 이거나 const char * 일 때는 template 함수 생성하지 말고 이걸 사용하라는 뜻.

536-539p

```
int main(void)
{
        cout << Max(11, 15) << endl;
        cout << Max('T', 'Q') << endl;
        cout << Max(3.5, 7.5) << endl;
        cout << Max("Simple", "Best") << endl;

        char str1[] = "Simple";
        char str2[] = "Best";
        cout << Max(str1, str2) << endl;
        return 0;
}
```

```
template <>
const char* Max<const char *>(const char* a, const char* b)
함수를 호출하게 됨.
```

```
template <>
char* Max<char *>(char* a, char* b)
함수를 호출하게 됨
```

[실행결과]

```
T Max(T a, T b)
15
T Max(T a, T b)
T
T Max(T a, T b)
7.5
const char* Max<const char*>(const char* a, const char* b)
Simple
char* Max<char*>(char* a, char* b)
Simple
```

# 클래스 템플릿
## (PointClassTemplate.cpp)

540-542p

```
#include <iostream>
using namespace std;
template <typename T>
class Point {
private:
        T xpos, ypos;
public:
        Point(T x = 0, T y = 0) : xpos(x), ypos(y) {  }
        void ShowPosition() const {
                cout << '[' << xpos << ", " << ypos << ']' << endl;
        }
};
int main(void) {
        Point<int> pos1(3, 4);
        pos1.ShowPosition();
        Point<double> pos2(2.4, 3.6);
        pos2.ShowPosition();
        Point<char> pos3('P', 'F');
        pos3.ShowPosition();
        return 0;
}
```

클래스 선언에도 template사용 가능.

Point 클래스의 경우, 좌표 정보가 정수, 실수, 문자로 다양화 될 수 있는데 이들 각각에 대하여 하나씩 클래스 선언하지 않고 클래스 template 만들어 주자.

컴파일러가 타입에 따라 template 클래스를 3개 만들게 된다.
단, template class 객체 선언시에는 타입 생략 불가!

하나의 .h 파일에 클래스 template 함수 정의까지 모두 포함시키자 :
이를 사용하는 곳에서는 함수 정의까지 알아야 하므로.

# 클래스 템플릿의 선언과 정의 분리
## (PointClassTemplateFuncDef.cpp)

543-544p

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

template <typename T>
class Point {
private:
        T xpos, ypos;
public:
        Point(T x = 0, T y = 0);
        void ShowPosition() const;
};

template <typename T>
Point<T>::Point(T x, T y) : xpos(x), ypos(y) {  }

template <typename T>
void Point<T>::ShowPosition() const {
        cout << '[' << xpos << ", " << ypos << ']' << endl;
}
```

[실행결과]
[3, 4]
[2.4, 3.6]
[P, F]

main 함수는 이전 예제와 동일!

문자 T 가 무슨 의미인지 각 함수 정의할 때마다 알려줘야 함.

546-547p

# 클래스 템플릿의 선언과 정의 분리
# (PointClassTemplateFuncDef.cpp)

<교재 읽어보기>

Template class의 경우 :

함수 구현 부까지 모두 알아야 사용 가능 (클래스 객체 선언 가능)하므로
헤더 파일에 함수 주현까지 모두 넣도록 하자.

# 배열 클래스의 템플릿화

- 아래 3개 클래스 (타입만 다르고 기능은 같음)를 하나의 클래스 template으로 만들자.
  - class BoundCheckIntArray { … }
  - class BoundCheckArray { … }
  - class BoundCheckPointPtrArray { … }
- 하나의 배열클래스 만들어 다양한 타입의 배열을 만들어보자
  - int
  - Point
  - Point *

547-551p

# 배열 클래스의 템플릿화
## (Point.h)

```
#ifndef __POINT_H_
#define __POINT_H_

#include <iostream>
using namespace std;

class Point
{
private:
        int xpos, ypos;
public:
        Point(int x=0, int y=0);
        friend ostream& operator<<(ostream& os, const Point& pos);
};

#endif
```

배열 만들 수 있게 default constructor

# 배열 클래스의 템플릿화 (Point.cpp)

```
#include <iostream>
#include "Point.h"
using namespace std;

Point::Point(int x, int y)
: xpos(x), ypos(y)
{  }

ostream& operator<<(ostream& os, const Point& pos)
{
        os<<'['<<pos.xpos<<", "<<pos.ypos<<']'<<endl;
        return os;
}
```

# 객체의 저장을 위한 배열 클래스 (ArrayTemplate.h 원본)

463-465p

```
templete <typename T>
class BoundCheckArray {
private:
    Point* arr;
    int arrlen;
    BoundCheckArray(const BoundCheckArray& arr) { }
    BoundCheckArray& operator=(const BoundCheckArray& arr) { }
public:
    BoundCheckArray(int len) :arrlen(len) { arr = new Point[len]; }
    Point& operator[] (int idx) {
        if (idx < 0 || idx >= arrlen) {
            cout << "Array index out of bound exception" << endl;
            exit(1);
        }
        return arr[idx];
    }
    Point operator[] (int idx) const {
        if (idx < 0 || idx >= arrlen) {
            cout << "Array index out of bound exception" << endl;
            exit(1);
        }
        return arr[idx];
    }
    int GetArrLen() const { return arrlen; }
    ~BoundCheckArray() { delete[]arr; }
};
```

# 배열 클래스의 템플릿화
## (ArrayTemplate.h 1/3)

547-551p

```
#ifndef __ARRAY_TEMPLATE_H_
#define __ARRAY_TEMPLATE_H_

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

template <typename T>
class BoundCheckArray  {
private:
        T * arr;
        int arrlen;
        BoundCheckArray(const BoundCheckArray& arr) { }
        BoundCheckArray& operator=(const BoundCheckArray& arr) { }
public:
        BoundCheckArray(int len);
        T& operator[] (int idx);
        T operator[] (int idx) const;
        int GetArrLen() const;
        ~BoundCheckArray();
};
```

547-551p

# 배열 클래스의 템플릿화
## (ArrayTemplate.h 2/3)

```
template <typename T>
BoundCheckArray<T>::BoundCheckArray(int len) :arrlen(len)
{
        arr=new T[len];
}

template <typename T>
T& BoundCheckArray<T>::operator[] (int idx)
{
        if(idx<0 || idx>=arrlen)
        {
                cout<<"Array index out of bound exception"<<endl;
                exit(1);
        }
        return arr[idx];
}
```

함수 정의를 분리해 놓으면 매 함수마다 template 선언 필수!

# 배열 클래스의 템플릿화
## (ArrayTemplate.h 3/3)

547-551p

```
template <typename T>
T BoundCheckArray<T>::operator[] (int idx) const {
        if(idx<0 || idx>=arrlen)
        {
                cout<<"Array index out of bound exception"<<endl;
                exit(1);
        }
        return arr[idx];
}

template <typename T>
int BoundCheckArray<T>::GetArrLen() const {
        return arrlen;
}

template <typename T>
BoundCheckArray<T>::~BoundCheckArray() {
        delete [] arr;
}
#endif
```

# 배열 클래스의 템플릿화
## (BoundArrayMain.cpp 1/2)

```
#include <iostream>
#include "ArrayTemplate.h"
#include "Point.h"
using namespace std;
int main(void) {
        /*** int형 정수 저장 ***/
        BoundCheckArray<int> iarr(5);

        for(int i=0; i<5; i++)
                iarr[i]=(i+1)*11;

        for(int i=0; i<5; i++)
                cout<<iarr[i]<<endl;


        /*** Point 객체 저장 ***/
        BoundCheckArray<Point> oarr(3);
        oarr[0]=Point(3, 4);
        oarr[1]=Point(5, 6);
        oarr[2]=Point(7, 8);
```

[실행결과]

```
11
22
33
44
55
[3, 4]
[5, 6]
[7, 8]
[3, 4]
[5, 6]
[7, 8]
```

# 배열 클래스의 템플릿화
# (BoundArrayMain.cpp 2/2)

547-551p

```
        for(int i=0; i<oarr.GetArrLen(); i++)
                cout<<oarr[i];

        /*** Point 객체의 주소 값 저장 ***/
        typedef Point * POINT_PTR;
        BoundCheckArray<POINT_PTR> parr(3);
        parr[0]=new Point(3, 4);
        parr[1]=new Point(5, 6);
        parr[2]=new Point(7, 8);

        for(int i=0; i<parr.GetArrLen(); i++)
                cout<<*(parr[i]);

        delete parr[0];
        delete parr[1];
        delete parr[2];

        return 0;
}
```

553-556p

# OOP 프로젝트 10단계

◆ 배열 클래스 BoundCheckAccountPtrArray 를 클래스 template 으로 대체하자

- 클래스 template 이름 : BoundCheckArray
  (ArrayTemplate.h -> BoundCheckArray.h)
  (방금 완성한 ArrayTemplate.h 내용 그대로 가져와서 사용.)
- AccountArray.h, AccountArray.cpp 파일 삭제.
- AccountHandler.h 수정

553-556p

# OOP 프로젝트 10단계
## (AccountHandler.h 수정)

```
#ifndef __ACCOUN_HANDLER_H__
#define __ACCOUN_HANDLER_H__

#include "Account.h"

#include "BoundCheckArray.h"

class AccountHandler
{
private:

        BoundCheckArray<Account*> accArr;

        int accNum;
        ...
        ...
```

실습
vector.cpp

# C++ STL container
## (vector 사용 예)

```
// 임의 순서의 숫자들을 vector에 넣어 보자. (break point로써 값을 확인)
#include <iostream>
#include <vector>
using namespace std;

int main(void) {
        int values[] = { 12, 6, 7 };
        vector<int> v1;
        for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++)
                v1.push_back(values[i]);
        return 0;
}
```

# C++ STL container
## (priority_queue)

pop() 할 때
max 또는 min
값을 꺼내 줌.

- **기본 형태**
  - priority_queue<T, Container, Compare>
- **우선순위 큐 인터페이스**
  - push/pop/top 연산 제공(높은 우선순위 요소가 위에 존재)
  - 추가 삭제
- **힙을 사용하여 구현됨 (요소들은 비교를 지원해야 함)**

실습
priority_queue.cpp

# C++ STL container (priority_queue)

pop() 할 때 max 또는 min 값을 꺼내 줌.

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
        int values[] = { 5,2,8,9,1,14 };
        priority_queue< int, vector<int>, less<int> > pq;
        cout << "push: ";
        for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++) {
                cout << values[i] << " ";
                pq.push(values[i]);
        }
        cout << endl;
        cout << "size: " << pq.size() << endl;
        cout << "pop: " << endl;
        while (!pq.empty()) {
                cout << "top: " << pq.top() << endl;
                pq.pop();
        }
        cout << endl;
        return 0;

}
```

비교연산 class 사용.
less : 작은 것부터
greater : 큰 것부터

Push elements

Pop elements

[실행결과]

```
push: 5 2 8 9 1 14
size: 6
pop:
top: 14
top: 9
top: 8
top: 5
top: 2
top: 1
```