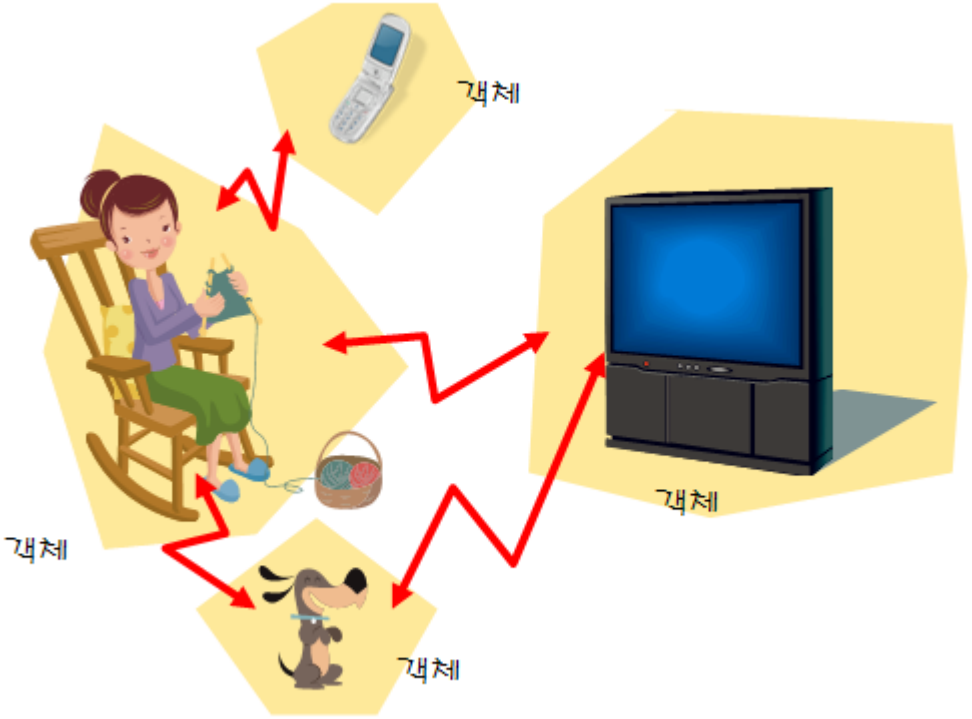
C++ Espresso

제12_1장 템플릿



이번 장에서 학습할 내용

- 함수 템플릿

일반적인
하나의 코드로
다양한
자료형을
처리하는
기법을
살펴봅시다.

복습

1. 예외 처리와 관련된 C++키워드가 아닌 것은? 3

1) try 2) catch 3) except 4) throw

2. 하나의 try{} 블록에 연결되는 catch(){}블록에 대한 설명으로 옳은 것은? 1

1) 여러 개 만들 수 있다
2) 있어도 되고 없어도 된다
3) 반드시 1개만 있어야 한다
4) 개발자가 catch(){}블록을 지정하지 않으면 디폴트 catch(){}블록이 만들어진다

복습

실행결과는? 4

```
try {
    throw 3;
}
catch (int x) {
    try {
        throw x + 1;
        cout << x;
    }
    catch (int y) {
        cout << y;
    }
}
```

함수 중복의 약점 – 중복 함수의 코드 중복

```
int main() {
  int a=4, b=5;
  myswap(a, b);
  cout << a << '\t' << b << endl;

  double c=0.3, d=12.5;
  myswap(c, d);
  cout << c << '\t' << d << endl;
}
```

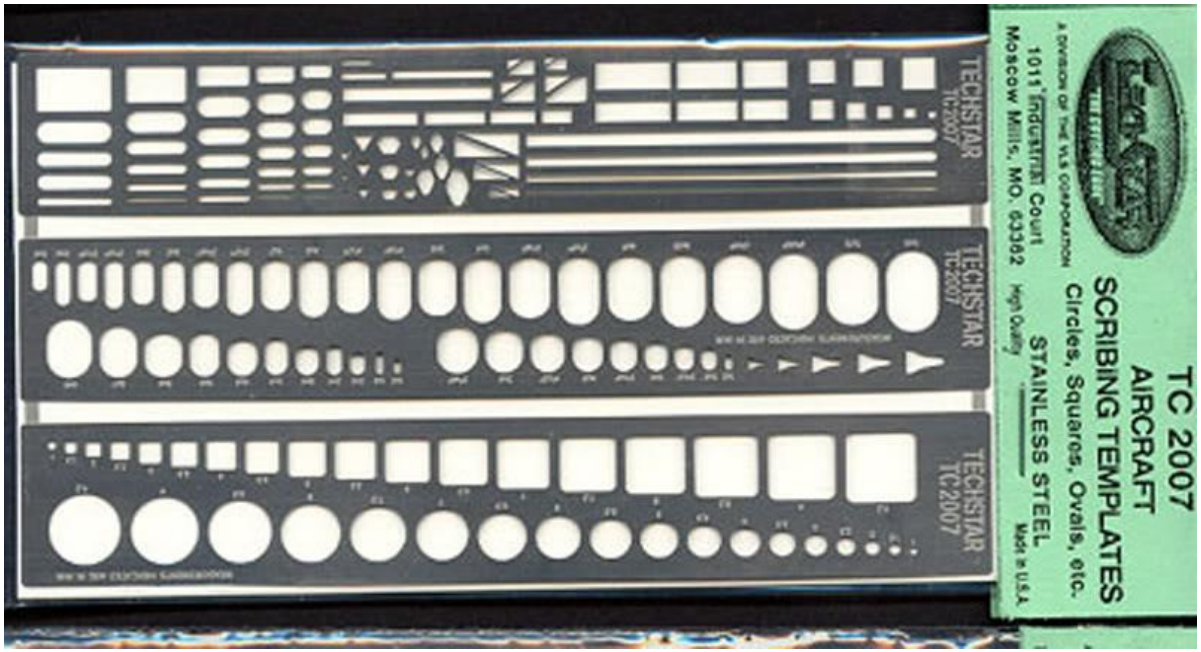
```
5       4
12.5    0.3
```

```
void myswap(int& a, int& b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
void myswap(double & a, double & b)
{
  double tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

동일한 코드
중복 작성

템플릿이란?

- 템플릿(template): 물건을 만들 때 사용되는 틀이나 모형을 의미
- 함수 템플릿(function template): 함수를 찍어내기 위한 형틀



함수 get_max()

```
int get_max(int x, int y)
{
   if( x > y ) return x;
   else return y;
}
```

함수 get_max()

```
float get_max(float x, float y)
{
   if( x > y ) return x;
   else return y;
}
```

핵심적인 내용은 같고 매개 변수의 타입만 달라진다.

일반화 프로그래밍

- 일반화 프로그래밍(generic programming): 일반적인 코드를 작성하고 이 코드를 정수나 문자열과 같은 다양한 타입의 객체에 대하여 재사용하는 프로그래밍 기법

템플릿 함수

```
____ get_max(___x , ___ y)
{
  if( x > y)   return x;
  else return y;
}
```

```
int get_max(int  x , int y)
{
  if( x > y)   return x;
  else return y;
}
```

일반화와 템플릿

- **템플릿**
  - 함수나 클래스를 일반화하는 C++ 도구
  - template 키워드로 함수나 클래스 선언
    - 변수나 매개 변수의 타입만 다르고, 코드 부분이 동일한 함수를 일반화시킴
  - 제네릭 타입 - 일반화를 위한 데이터 타입

- **템플릿 선언**

```
template <class T> 또는
template <typename T>

2 개의 제네릭 타입을 가진 템플릿 선언
template <class T1, class T2>
```

템플릿을 선언하는 키워드

제네릭 타입을 선언하는 키워드

제네릭 타입 T 선언

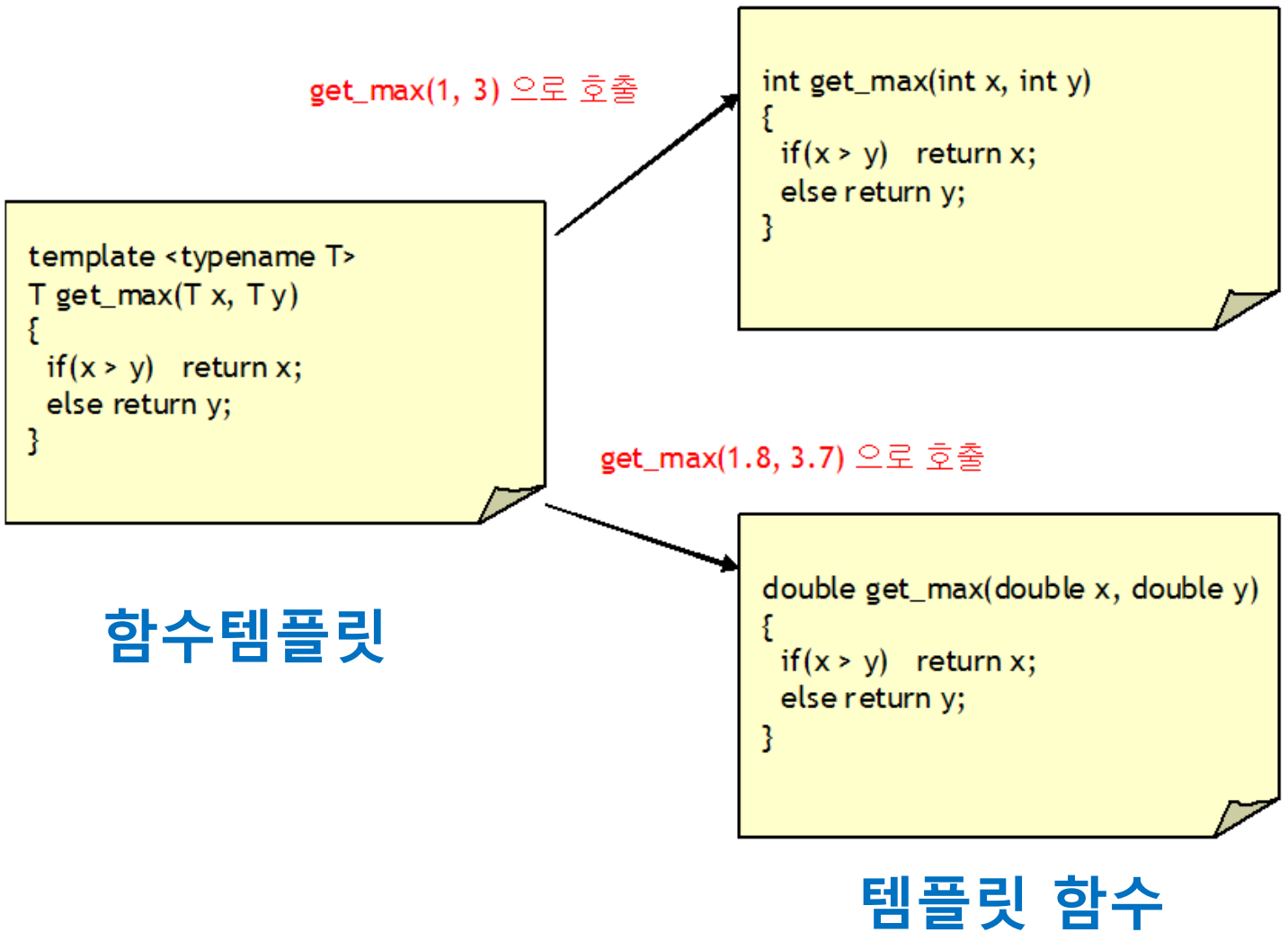
```
template <class T>
void myswap (T & a, T & b) {
  T tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

템플릿을 이용한 제네릭 함수 **myswap**

get_max()

```
template<typename T>
T get_max(T x, T y)
{
   if( x > y ) return x;
   else return y;
}
```

템플릿 함수의 사용



중복 함수들로부터 템플릿 만들기 사례

```
void myswap(int & a, int & b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

```
void myswap (double & a, double & b)
{
  double tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

중복 함수들

(일반화)

```
template <typename T>
void myswap (T & a, T & b)
{
  T tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

템플릿을 이용한
제네릭 함수

예제

- 다음과 같이 get_max 함수가 중복되어 작성된 코드를 템플릿을 이용하여 수정해보자

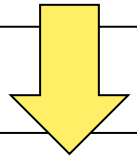
```
int main()
{
    cout << get_max<int>(1, 3) << endl;
    cout << get_max<double>(1.2, 3.9) << endl;
    return 0;
}
```

```
C:₩Windows₩system32₩cmd.exe
3
3.9
계속하려면 아무 키나 누르십시오 . . .
```

```
template<typename T>
T get_max(T x, T y){
    if (x > y) return x;
    else return y;
}
```

보충

```
int main()
{
    cout << get_max<int>(1, 3) << endl;
    cout << get_max<double>(1.2, 3.9) << endl;
    return 0;
}
```



```
int main()
{
    cout << get_max(1, 3) << endl;
    cout << get_max(1.2, 3.9) << endl;
    return 0;
}
```

템플릿으로부터의 구체화

- 구체화(specialization)
  - 템플릿의 제네릭 타입에 구체적인 타입 지정
    - 템플릿 함수로부터 구체화된 함수의 소스 코드 생성(컴파일시)

```
template <class T>
void myswap(T & a,
T & b) {
  T tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

구체화

```
void myswap(int & a, int & b)
{
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a=4, b=5;
  myswap(a, b);
}
```

호출

```
int main() {
  int a=4, b=5;
  myswap(a, b);
}
```

myswap(a, b) 호출에 필요한 함수 구체화

제네릭 함수로부터 구체화된 함수 생성 사례

```
template <class T>
void myswap(T & a, T &
b) {
   T tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

제네릭 함수

int a=4, b=5;
myswap(a, b);

T -> int

double c=0.3,
d=12.5;
myswap(c, d);

T ->double

char e='a', f='k';
myswap(e, f);

T ->
char

구체화

```
void myswap(int & a, int & b) {
   int tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

```
void myswap(double & a, double &
b) {
   double tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

```
void myswap(char & a, char & b) {
   char tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

구체화된 버전의 C++ 소스 생성

템플릿 장점과 제네릭 프로그래밍

- **템플릿 장점**
  - 함수의 재사용
    - 높은 소프트웨어의 생산성과 유용성
- **템플릿 단점**
  - 컴파일러에 따라 지원하지 않을 수 있음
  - 컴파일 오류 메시지 빈약, 디버깅에 많은 어려움
- **제네릭 프로그래밍**
  - generic programming
    - 일반화 프로그램이라고도 부름
    - 제네릭 함수나 제네릭 클래스를 활용하는 프로그래밍 기법

둘 이상의 형에 대한 템플릿 선언

```
template <typename T1, typename T2>
void ShowData(double num){
    cout << (T1)num << ", " << (T2)num << endl;
}
int main()
{
    ShowData<char, int>(65);
    ShowData<char, int>(67);
    ShowData<char, double>(68.9);
    ShowData<int, double>(70.4);
    return 0;
}
```

```
A, 65
C, 67
D, 68.9
70, 70.4
계속하려면 아무 키
```

배열의 합을 구하여 리턴하는 제네릭 add() 함수 만들기

- 배열과 크기를 매개 변수로 받아 합을 구하여 리턴하는 제네릭 함수 add()를 작성하라.

```
int main(){
    int x[] = { 1, 2, 3, 4, 5 };
    double d[] = { 1.2, 2.3, 3.4, 4.5, 5.6, 6.7 };

    cout << "sum of x[] = " << add(x, 5) << endl;
    cout << "sum of x[] = " << add(d, 6) << endl;
}
```

정답

```
template <class T>
T add(T data[], int n)
{
    T sum = 0;
    for (int i = 0; i < n; i++){
        sum += data[i];
    }
    return sum;
}
```

Q & A