

제14장 예외 처리와 템플릿

1. 예외 처리의 개요를 학습한다.
2. 예외 처리를 적용할 수 있다.
3. 템플릿의 개념을 이해한다.
4. 템플릿으로 간단한 클래스를 설계할 수 있다.

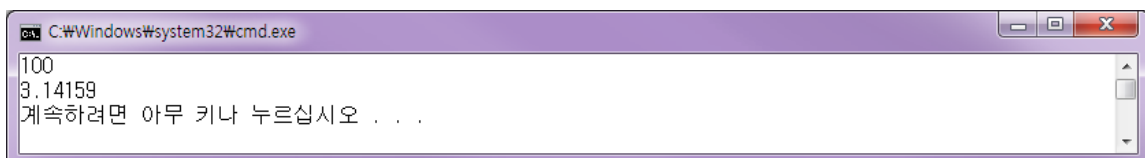
1

이번 장에서 만들어 볼 프로그램

(1) 한사람이 먹을 수 있는 피자 조각 개수를 계산하는 프로그램을 작성하자. 사람 수를 0으로 입력하여도 오류가 일어나지 않게 한다.



(2) 템플릿을 사용하여 어떤 자료형도 저장할 수 있는 Box 클래스를 작성하고 실험해 보자.



2

14.2 예외 처리란?

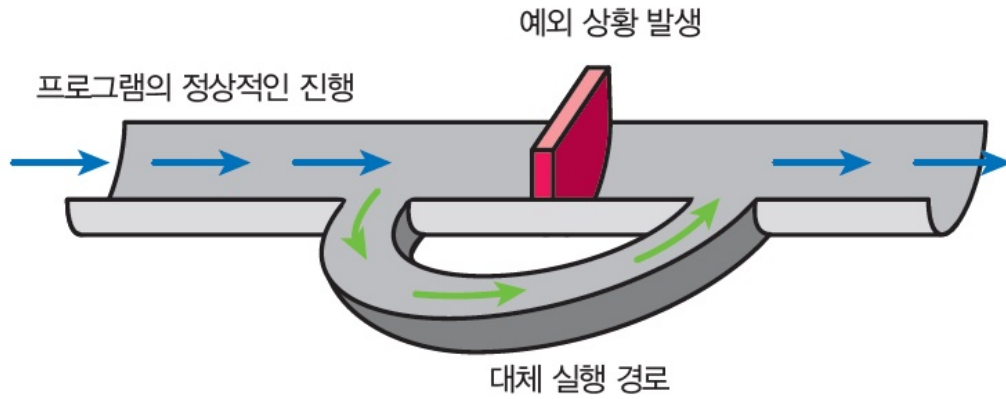
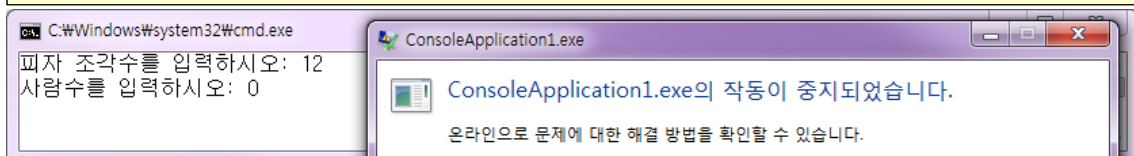


그림 14.1 예외 처리의 개요

3

예제

```
int main()
{
    int pizza_slices = 0;
    int persons = -1;
    int slices_per_person = 0;
    cout << "피자 조각수를 입력하십시오: ";
    cin >> pizza_slices;
    cout << "사람수를 입력하십시오: ";
    cin >> persons;
    slices_per_person = pizza_slices / persons;
    cout << "한사람당 피자는 " << slices_per_person << "입니다." << endl;
    return 0;
}
```



4

14.3 예외 처리기

문법 13.1

예외 처리

```
try {  
    // 예외가 발생할 수 있는 코드  
    if(예외가 발생하면)  
        throw exception;  
} catch (예외타입 매개변수) {  
    // 예외를 처리하는 코드  
}
```

5

try-catch 블록

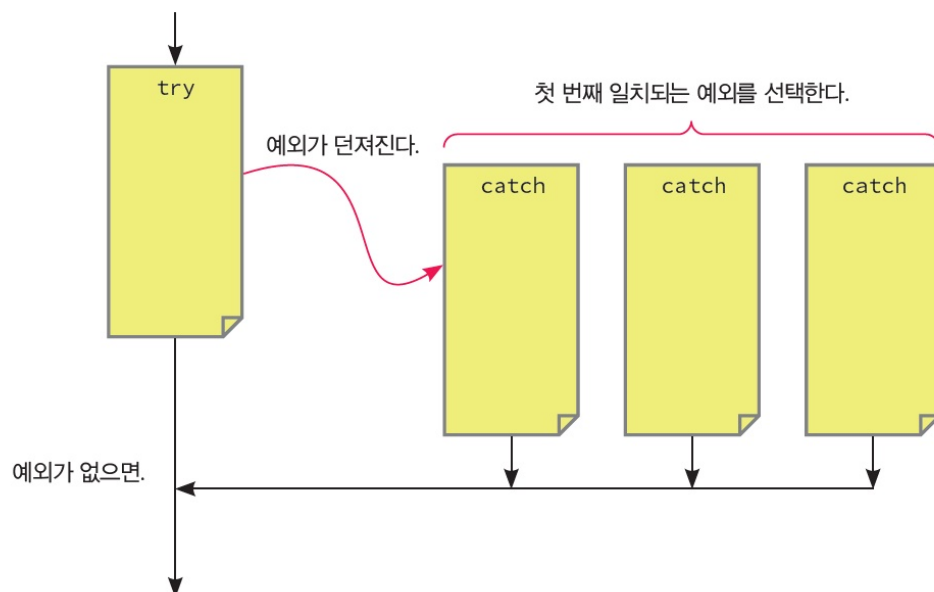


그림 14.2 try블록은 예외가 발생할 수 있는 위험한 코드이다. catch 블록은 예외를 처리하는 코드이다.

6

예제

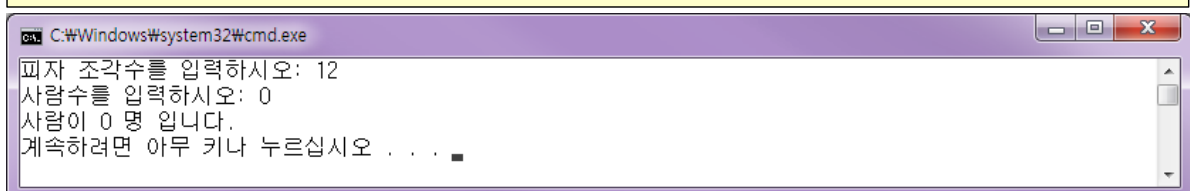
```
int main()
{
    int pizza_slices = 0;
    int persons = -1;
    int slices_per_person = 0;

    try
    {
        cout << "피자 조각수를 입력하십시오: ";
        cin >> pizza_slices;
        cout << "사람수를 입력하십시오: ";
        cin >> persons;
```

7

예제

```
        if (persons == 0)
            throw persons;
        slices_per_person = pizza_slices / persons;
        cout << "한사람당 피자는 " << slices_per_person << "입니다." << endl;
    }
    catch (int e)
    {
        cout << "사람이 " << e << " 명 입니다. " << endl;
    }
    return 0;
}
```



8

try/catch 블록에서의 실행 흐름

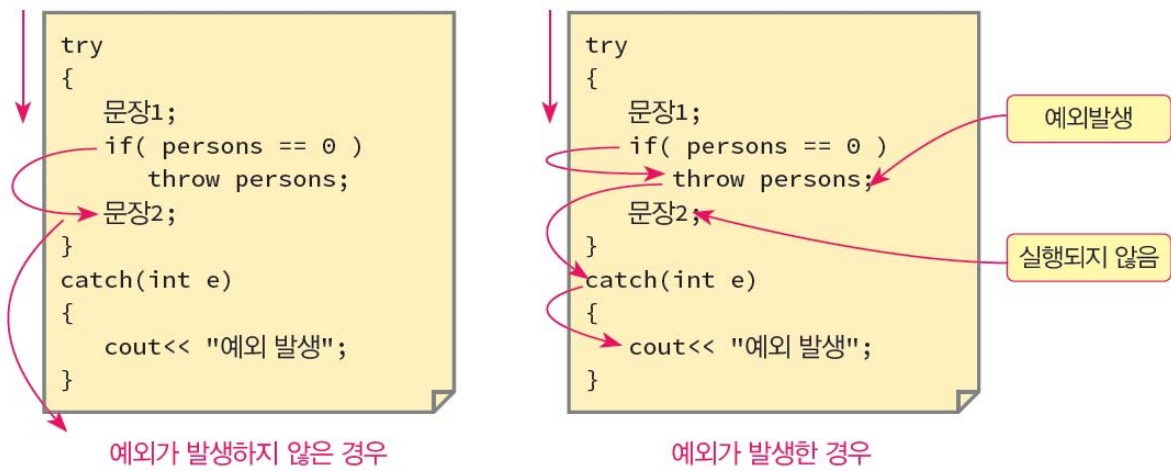


그림 14.3 try/catch 블록에서의 실행 흐름

9

catch 블록의 매개 변수



예외 처리기의 매개 변수

그림 14.4 catch 블록의 매개 변수

10

타입이 일치되는 예외만 처리된다.

```
try {
    int person = 0;
    ...
    if (persons == 0)
        throw persons;
    ...
}
catch(char e)
{
    cout << "사람이 " << e << " 명 입니다. " << endl;
}
```

11

14.4 예외 전달

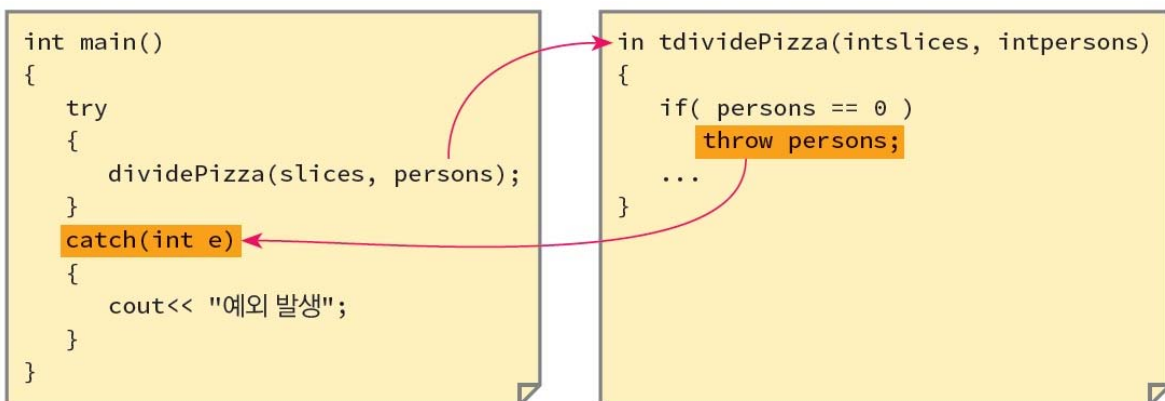


그림 14.5 예외는 함수를 넘어서 전달될 수 있다.

12

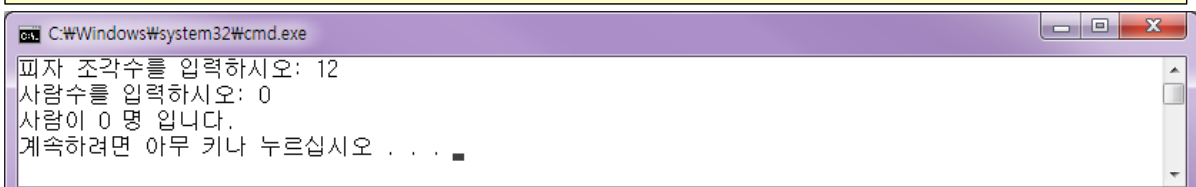
예제

```
#include <iostream>
using namespace std;
int dividePizza(int pizza_slices, int persons);
int main()
{
    int pizza_slices = 0;
    int persons = 0;
    int slices_per_person = 0;
    try
    {
        cout << "피자 조각수를 입력하시오: ";
        cin >> pizza_slices;
        cout << "사람수를 입력하시오: ";
        cin >> persons;
        slices_per_person = dividePizza(pizza_slices, persons);
        cout << "한사람당 피자는 " << slices_per_person << "입니다." << endl;
    }
}
```

13

예제

```
        catch (int e)
        {
            cout << "사람이 " << e << " 명 입니다. " << endl;
        }
        return 0;
    }
    int dividePizza(int pizza_slices, int persons)
    {
        if (persons == 0)
            throw persons;
        return pizza_slices / persons;
    }
}
```

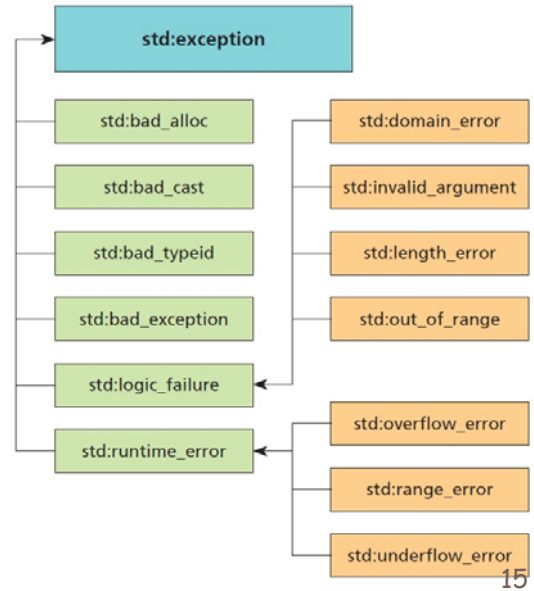


```
C:\Windows\system32\cmd.exe
피자 조각수를 입력하시오: 12
사람수를 입력하시오: 0
사람이 0 명 입니다.
계속하려면 아무 키나 누르십시오 . . .
```

14

표준 예외

- C++ 표준 라이브러리에서 예외가 발생하면 `std::exception`이라는 특별한 예외가 발생하며 `<exception>`헤더에 정의된다.



예제

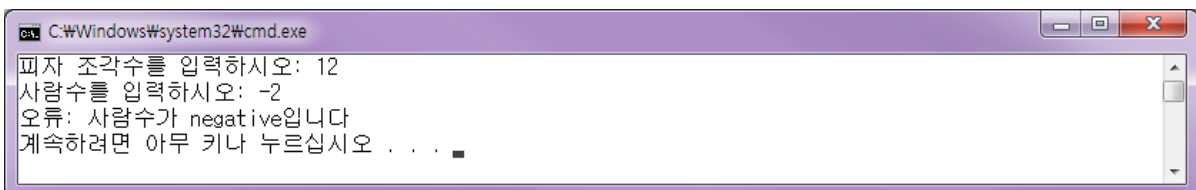
```
#include <iostream>
#include <exception>
using namespace std;
int main() {
    try {
        int* p = new int[100000];
        delete p;
    }
    catch (exception& e) {
        cout << "표준 예외가 발생했습니다. " << e.what() << endl;
    }
    return 0;
}
```


14.5 다중 catch 문장

```
try {
    cout << "피자 조각수를 입력하시오: ";
    cin >> pizza_slices;
    cout << "사람수를 입력하시오: ";
    cin >> persons;
    if (persons < 0) throw "negative";           // 예외 발생!
    if (persons == 0) throw persons;            // 예외 발생!
    slices_per_person = pizza_slices / persons;
    cout << "한사람당 피자는 " << slices_per_person << "입니다." << endl;
}
catch (const char *e) {
    cout << "오류: 사람수가 " << e << "입니다" << endl;
}
catch (int e) {
    cout << "오류: 사람이 " << e << " 명입니다." << endl;
}
```

17

실행결과



```
C:\Windows\system32\cmd.exe
피자 조각수를 입력하시오: 12
사람수를 입력하시오: -2
오류: 사람수가 negative입니다
계속하려면 아무 키나 누르십시오 . . . .
```

18

구체적인 예외를 먼저 잡는다.

```
try {  
    getInput();  
}  
catch(TooSmallException e) {  
    //TooSmallException만 잡힌다.  
}  
catch(...) {  
    //TooSmallException을 제외한 나머지 예외들이 잡힌다.  
}
```

19

14.6 함수 템플릿

- C++에서도 하나의 형틀을 만들어서 다양한 코드를 생산해 내도록 할 수 있는데 이것을 템플릿이라고 한다. 함수 템플릿(function template)은 함수를 찍어내기 위한 형틀이다.

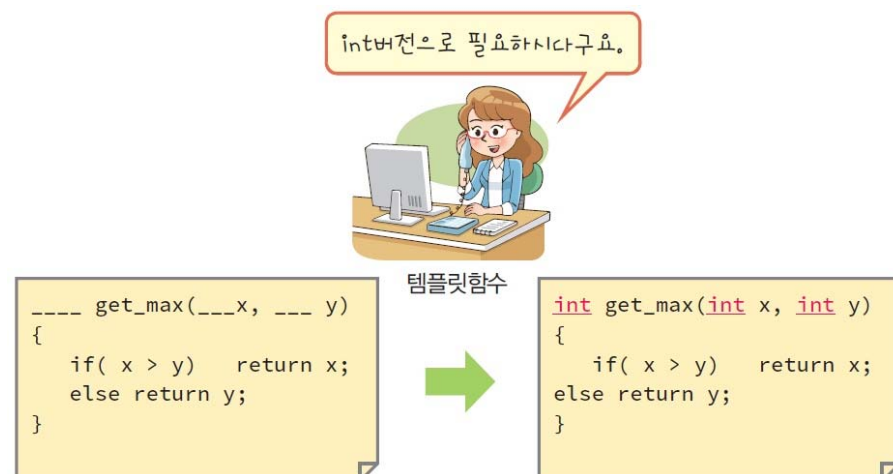


그림 14.6 일반화 프로그래밍의 개념

20

일반화 프로그래밍(generic programming)

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

21

일반화 프로그래밍(generic programming)

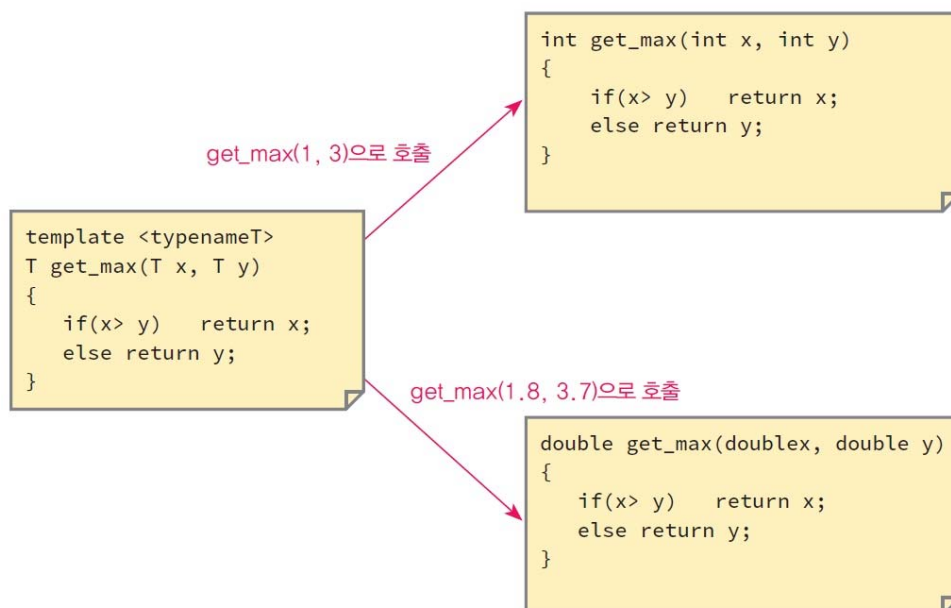


그림 14.7 템플릿 함수

22

예제

```
#include <iostream>
using namespace std;
template <typename T>
T get_max(T x, T y)
{
    if (x > y) return x;
    else return y;
}
int main()
{
    cout << get_max(1, 3) << endl;
    cout << get_max(1.2, 3.9) << endl;
    return 0;
}
```



23

함수 템플릿의 특수화(Specialization): 도입

실행 NeedSpecialFunctionTemplate.cpp -> SpecialFunctionTemplate.cpp

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

대소비교 함수 템플릿! 큰 값을 반환한다.

```
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;
    return 0;
}
```

정수, 실수 그리고 문자를 대상으로는 Max 함수의 호출의 의미를 갖는다. 그러나 문자열을 대상으로 호출이 되면 의미를 갖지 않는다!

```
const char* Max(const char* a, const char* b)
{
    return strlen(a) > strlen(b) ? a : b ;
}
```

문자열의 길이비교가 목적인 경우 어울리는 함수!

일반적인 상황에서는 Max 템플릿 함수가 호출되고, 문자열이 전달되는 경우에는 문자열의 길이를 비교하는 Max 함수를 호출하게 할 수 있을까? → 함수 템플릿의 특수화 등장 배경

24

함수 템플릿의 특수화(Specialization): 적용

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

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

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;
}
```

함수 템플릿 Max를
char * 형에 대해서 특수화

함수 템플릿 Max를
const char * 형에 대해서 특수화

실행결과

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

25

함수 템플릿의 특수화(Specialization): 비교

```
template <>
char* Max(char* a, char* b)
{ .... }

template <>
const char* Max(const char* a, const char* b)
{ .... }
```

특수화하는 자료형의 정보가 생략된 형태

```
template <>
char* Max<char*>(char* a, char* b)
{ .... }

template <>
const char* Max<const char*>(const char* a, const char* b)
{ .... }
```

특수화하는 자료형의 정보를 명시한 형태

26

함수 템플릿과 함수 오버로딩

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

template <typename T>
void swap_values(T& x, T& y)
{
    T temp;
    temp = x;
    x = y;
    y = temp;
}
```

27

함수 템플릿과 함수 오버로딩

```
void swap_values(char* s1, char* s2)
{
    int len;

    len = (strlen(s1) >= strlen(s2)) ? strlen(s1) : strlen(s2);
    char* tmp = new char[len + 1];

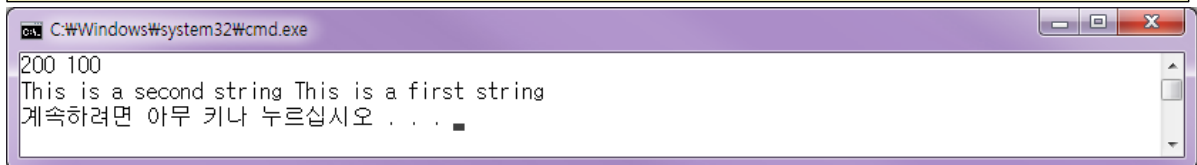
    strcpy(tmp, s1);
    strcpy(s1, s2);
    strcpy(s2, tmp);
    delete[] tmp;
}
```

28

함수 템플릿과 함수 오버로딩

```
int main()
{
    int x = 100, y = 200;
    swap_values(x, y);      // x, y가 모두 int 타입 - OK!
    cout << x << " " << y << endl;

    char s1[100] = "This is a first string";
    char s2[100] = "This is a second string";
    swap_values(s1, s2);    // s1, s2가 모두 배열 - 오버로딩 함수 호출
    cout << s1 << " " << s2 << endl;
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
200 100
This is a second string This is a first string
계속하려면 아무 키나 누르십시오 . . .
```

29

두개의 타입 매개 변수를 갖는 함수 템플릿

```
template<typename T1, typename T2>
void copy(T1 a1[], T2 a2[], int n)
{
    for (int i = 0; i < n; ++i)
        a1[i] = a2[i];
}
```

30

예제 #1

```
template<typename T>
void copy_array(T a[], T b[], int n)
{
    for (int i = 0; i < n; ++i)
        a[i] = b[i];
}
```

31

예제 #2

```
template <typename T>
T get_first(T a[])
{
    return a[0];
}
```

32

-
- 변수의 절대값을 구하는 `int abs(int x)`를 템플릿 함수로 정의하라.
 - 두 수의 합을 계산하는 `int add(int a, int b)`를 템플릿 함수로 구현하라.
 - `displayArray()`라는 메소드는 배열을 매개 변수로 받아서 반복 루프를 사용하여 배열의 원소를 화면에 출력한다. 어떤 타입의 배열도 처리할 수 있도록 제네릭 메소드로 정의하라.

문제 [함수 템플릿의 정의]

-
- 인자로 전달되는 두 변수에 저장된 값을 서로 교환하는 `SwapData`라는 이름의 함수를 템플릿으로 정의해보자. 값의 교환이 이뤄짐을 확인할 수 있도록 `main` 함수를 구성해보라.

문제 [함수 템플릿의 정의]

- 다음은 `int`형 배열에 저장된 값을 모두 더해서 그 결과를 반환하는 기능의 함수이다. 이 함수를 **템플릿**을 사용하여 정의하고, 다양한 자료형의 배열을 대상으로 합을 계산하는 예제를 작성해보라.

```
int SumArray(int arr[], int len)
{
    int sum=0;
    for(int i=0; i<len; i++)
        sum+=arr[i];
    return sum;
}
```

35

14.7 클래스 템플릿

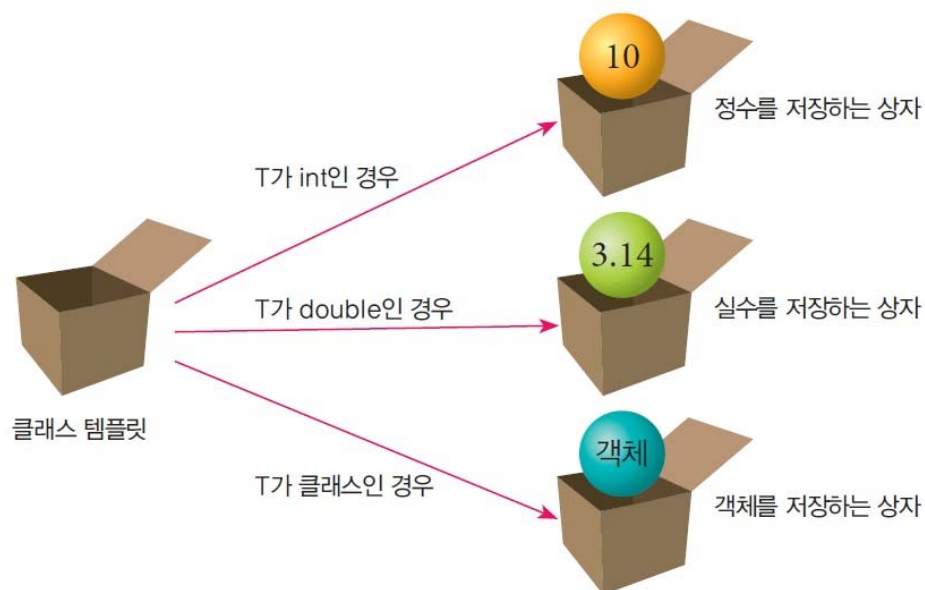


그림 14.9 클래스 템플릿

36

클래스 템플릿

문법 13.2

클래스 템플릿

```
template <typename T>
class 클래스이름
{
    ...// T를 어디서든지 사용할 수 있다.
}
```

함수 템플릿과 달리 인스턴스를 선언할 때는 **typename**을 반드시 적어야 함

37

예제

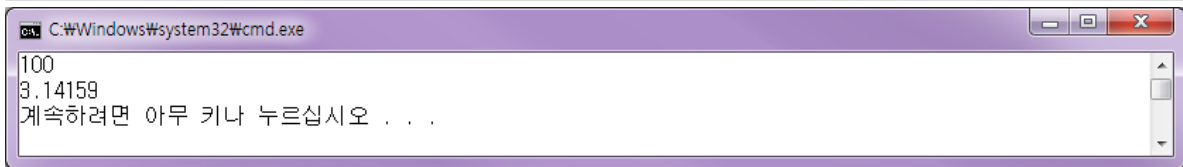
```
#include <iostream>
using namespace std;
template <typename T>
class Box {
    T data; // T는 타입(type)을 나타낸다.
public:
    Box() { }
    void set(T value) {
        data = value;
    }
    T get() {
        return data;
    }
};
```

38

예제

```
int main()
{
    Box<int> box;
    box.set(100);
    cout << box.get() << endl;

    Box<double> box1;
    box1.set(3.141592);
    cout << box1.get() << endl;
    return 0;
}
```



39

두개 이상의 타입 매개 변수를 가지는 경우



그림 14.10 Box2 클래스 템플릿

40

예제

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
class Box2 {
    T1 first_data; // T1은 타입(type)을 나타낸다.
    T2 second_data; // T2는 타입(type)을 나타낸다.
public:
    Box2() { }
    T1 get_first();
    T2 get_second();
    void set_first(T1 value) {
        first_data = value;
    }
    void set_second(T2 value) {
        second_data = value;
    }
};
```

41

예제

```
template <typename T1, typename T2>
T1 Box2<T1, T2>::get_first() {
    return first_data;
}
template <typename T1, typename T2>
T2 Box2<T1, T2>::get_second() {
    return second_data;
}
int main()
{
    Box2<int, double> b;
    b.set_first(10);
    b.set_second(3.14);
    cout << "(" << b.get_first() << ", " << b.get_second() << ")" << endl;
    return 0;
}
```



42

멤버 선언 및 정의

- 클래스 템플릿의 선언과 정의를 분리 가능
 - ▣ 분리된 정의에 `template<typename T>`을 매번 선언해야 함
 - ▣ `TemplateMember.cpp`
 - 멤버의 정의를 클래스 선언 밖으로 빼내면
 - 반드시 클래스 이름에 이어 `<type>`을 기술해야 함
- ```
template <typename T>
T CTest<T>::TestFunc()
```

43

### 클래스 템플릿의 선언과 정의의 분리

PointClassTemplateFuncDef.cpp

```
template <typename T>
class SimpleTemplate
{
public:
 T SimpleFunc(const T& ref);
};
```

함수의 선언

```
template <typename T>
T SimpleTemplate<T>::SimpleFunc(const T& ref)
{

}
```

템플릿 외부의 함수정의

`SimpleTemplate :: SimpleFunc( . . . )`

✓ `SimpleTemplate` 클래스의 멤버함수 `SimpleFunc`를 의미함

`SimpleTemplate<T> :: SimpleFunc( . . . )`

✓ `T`에 대해서 템플릿으로 정의된 `SimpleTemplate`의 멤버함수 `SimpleFunc`를 의미함

`template <typename T>`

✓ `<T>`가 들어가면 이 `T`가 의미하는 바를 항상 설명해야 한다.

44

## 헤더파일과 소스파일의 구분

실행 PointTemplate.h, PointTemplate.cpp, PointMain.cpp

```
#ifndef _POINT_TEMPLATE_H_
#define _POINT_TEMPLATE_H_

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

헤더파일

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

소스파일 1

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

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;
}
```

소스파일 2

기준에 의해서 헤더파일과 소스파일을 잘 구분하였다. 그러나 컴파일 오류가 발생한다! 이유는?

45

## 파일을 나눌 때에는 고려할 사항이 있습니다.

```
#ifndef _POINT_TEMPLATE_H_
#define _POINT_TEMPLATE_H_

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

헤더파일

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

소스파일 1

```
#include <iostream>
#include "PointTemplate.h"
#include "PointTemplate.cpp"
using namespace std;
int main(void)
{

 return 0;
}
```

위의 소스파일을 컴파일 할 때 Point<int>, Point<double>, Point<char> 템플릿 클래스가 만들어져야 한다. 따라서 Point 클래스 템플릿의 정의부에 대한 정보도 필요로 한다.

해결책 1. 클래스 템플릿의 정의부를 담고 있는 소스파일을 포함시킨다.

해결책 2. 헤더파일에 클래스 템플릿의 정의부를 포함시킨다.

46

## 템플릿과 STATIC

### 함수 템플릿과 static 지역변수 실행 FunctionTemplateStaticVar.cpp

```
template <typename T>
void ShowStaticValue(void)
{
 static T num=0;
 num+=1;
 cout<<num<<" ";
}

void ShowStaticValue<int>(void)
{
 static int num=0;
 num+=1;
 cout<<num<<" ";
}

void ShowStaticValue<long>(void)
{
 static long num=0;
 num+=1;
 cout<<num<<" ";
}

int main(void)
{
 ShowStaticValue<int>();
 ShowStaticValue<int>();
 ShowStaticValue<int>();
 cout<<endl;
 ShowStaticValue<long>();
 ShowStaticValue<long>();
 ShowStaticValue<long>();
 cout<<endl;
 ShowStaticValue<double>();
 ShowStaticValue<double>();
 ShowStaticValue<double>();
 return 0;
}
```

함수 템플릿의 static 변수는 템플릿 함수 별로 독립적이다!

```
1 2 3
1 2 3
1 2 3
```

실행결과



## 클래스 템플릿과 static 멤버변수

### ClassTemplateStaticMem.cpp

```
template <typename T>
class SimpleStaticMem
{
private:
 static T mem;
public:
 void AddMem(int num) { mem+=num; }
 void ShowMem() { cout<<mem<<endl; }
};

template <typename T>
T SimpleStaticMem<T>::mem=0; // 이는 템플릿 기반의 static 멤버 초기화 문장이다.
```

클래스 템플릿의 **static** 변수는  
템플릿 클래스 별로 독립적이다!  
따라서 템플릿 클래스 별 객체들  
사이에서만 공유가 이뤄진다.

```
class SimpleStaticMem<int>
{
private:
 static int mem;
public:
 void AddMem(int num) { mem+=num; }
 void ShowMem() { cout<<mem<<endl; }
};

int SimpleStaticMem<int>::mem=0;
```

SimpleStaticMem<int>의 mem은  
SimpleStaticMem<int>의 개체간 공유

```
class SimpleStaticMem<double>
{
private:
 static double mem;
public:
 void AddMem(double num) { mem+=num; }
 void ShowMem() { cout<<mem<<endl; }
};

double SimpleStaticMem<double>::mem=0;
```

SimpleStaticMem<double>의 mem은  
SimpleStaticMem<double>의 개체간 공유

49

## template<typename T> vs. template<>

```
template <typename T>
class SoSimple
{
public:
 T SimpleFunc(T num) { . . . }
};
```

템플릿임을 알리며 T가 무엇인지에 대한  
설명도 필요한 상황



```
template <typename T>
T SimpleStaticMem<T>::mem=0;
```

```
template <>
class SoSimple<int>
{
public:
 int SimpleFunc(int num) { . . . }
};
```

템플릿과 관련 있음을 알리기만 하면 되는 상황



```
template <>
long SimpleStaticMem<long>::mem=5;
```

**static 멤버 초기화의 특수화**

50

## 템플릿 매개변수

Template <typename T, typename T2>

- ▣ type을 여러 개 작성할 수 있다

Template <typename T, **int nSize**>

- ▣ type 중 일부는 구체적일 수 있음
- ▣ 템플릿 매개변수를 활용한 예: TemplateParam.cpp

```
template <typename T, int nSize>
```

```
class CMyArray ...
```

```
CMyArray<int, 3> arr;
```

- ▣ 디폴트 값 지정도 가능

```
template <typename T = int, int nSize = 3>
```

```
class CMyArray ...
```

```
CMyArray< > arr;
```

## 템플릿 매개변수에는 변수의 선언이 올 수 있습니다.

```
template <typename T, int len>
class SimpleArray
{
private:
 T arr[len];
public:
 T& operator[] (int idx)
 {
 return arr[idx];
 }
};
```



```
class SimpleArray<int, 5>
{
private:
 int arr[5];
public:
 int& operator[] (int idx) { return arr[idx]; }
};
```

**SimpleArray<int, 5> i5arr;**  
**SimpleArray<int, 5>형 템플릿 클래스**

템플릿의 인자로 변수의 선언이 올 수도 있다!



템플릿 인자를 통해서 **SimpleArray<int, 5>**와 **SimpleArray<int, 7>**이 서로 다른 자료형으로 인식되게 할 수 있다.  
 이로써 **SimpleArray<int, 5>**와 **SimpleArray<int, 7>**사이에서의 연계성을 완전히 제거할 수 있다.

```
int main(void)
{
 SimpleArray<int, 5> i5arr1;
 SimpleArray<int, 7> i7arr1;
 i5arr1=i7arr1; // 컴파일 Error!

}
```

```
class SimpleArray<double, 7>
{
private:
 double arr[7];
public:
 double& operator[] (int idx) { return arr[idx]; }
};
```

**SimpleArray<double, 7> i7arr;**  
**SimpleArray<double, 7>형 템플릿 클래스**

53

## 템플릿 매개변수는 디폴트 값 지정도 가능합니다.

TemplateParaDefaultValue.cpp

```
template <typename T=int, int len=7> 디폴트 값 지정 가능!
class SimpleArray
{
private:
 T arr[len];
public:
 T& operator[] (int idx) { return arr[idx]; }
 SimpleArray<T, len>& operator=(const SimpleArray<T, len> &ref)
 {
 for(int i=0; i<len; i++)
 arr[i]=ref.arr[i];
 }
};
```

```
int main(void)
{
 SimpleArray<> arr; T에 int, len에 7의 디폴트 값 지정!
 for(int i=0; i<7; i++)
 arr[i]=i+1;
 for(int i=0; i<7; i++)
 cout<<arr[i]<<" ";
 cout<<endl;
 return 0;
}
```

실행결과

1 2 3 4 5 6 7

54

## 템플릿 특수화

### 함수 템플릿 특수화

- 특별한 타입의 경우 전혀 다른 코드를 적용할 때
- TemplateFunction.cpp 실행

- ▣ 문자열과 다른 자료형을 나눠서 정의한 함수 템플릿

```
template <typename T>
```

```
T Add (T a, T b) { return a + b; }
```

```
template <>
```

```
char* Add(char *pszLeft, char * pszRight) { ... }
```

## 클래스 템플릿 특수화

```
template <typename T>
class SoSimple
{
public:
 T SimpleFunc(T num) { }
};
```



```
template <>
class SoSimple<int>
{
public:
 int SimpleFunc(int num) { }
};
```

SoSimple 클래스 템플릿에 대해서 int형에 대한 특수화

✓ 클래스 템플릿을 특수화하는 이유는 특정 자료형을 기반으로 생성된 객체에 대해, 구분이 되는 다른 행동양식을 적용하기 위함이다.

✓ 함수 템플릿을 특수화하는 방법과 이유, 그리고 클래스 템플릿을 특수화하는 방법과 이유는 동일하다.

**실행 ClassTemplateSpecialization.cpp**  
**TemplateSpecial.cpp**

57

## 클래스 템플릿의 부분 특수화

ClassTemplatePartialSpecialization.cpp

```
template <typename T1, typename T2>
class MySimple { }
```

MySimple 클래스 템플릿



```
template <>
class MySimple<char, int> { }
```

MySimple 클래스 템플릿의 <char, int>에  
대한 특수화



```
template <typename T1>
class MySimple<T1, int> { }
```

MySimple 클래스 템플릿의 <T1, int>에 대한  
부분적 특수화

T2가 int 인 경우에는 MySimple<T1, int>를  
대상으로 인스턴스가 생성된다.

위와 같이 <char, int>형으로 특수화, 그리고 <T1, int> 에 대해서 부분 특수화가  
모두 진행된 경우 특수화가 부분 특수화에 앞선다. 즉, <char, int>를 대상으로 객체  
생성시 특수화된 클래스의 객체가 생성된다.

58

# Q & A

---

