Chapter 11. 예외 처리와 템플릿

박 종 혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

11-1. 예외 처리

예외 처리(Exception Handling)

- 예외란?
  우리가 당연하다고 가정한 상황이 거짓이 되는 경우
  - 대표적 경우
    - 컴퓨터에 사용 가능한 메모리가 부족한 경우
    - 당연히 있을 것이라 생각한 파일이 없는 경우
    - 사용자가 잘못된 값을 입력하는 경우

- 예외 처리란?
  이러한 상황이 발생한 경우에도 프로그램이 올바르게 동작할 수 있도록 처리하는 것
  - 예) 컴퓨터에 사용 가능한 메모리가 부족해서 동적 메모리 할당이 실패한 경우
    --> 예외 처리를 잘 하지 못한 프로그램은 비정상 종료할 수 있음
    반면, 예외 처리를 잘 한 프로그램은 사용자가 작업 중이던 정보를 잘 보관한 후에 정상적으로 종료할 수 있음

1. 기존의 예외처리 방식

- 예외 처리
  - 일반적이지 않은 프로그램의 흐름의 처리
  - 에러가 아니다!

```
int main(void)
{
    int a, b;
    cout<<"두개의 숫자 입력 : ";
    cin>>a>>b;

    cout<<"a/b의 몫 : "<<a/b<<endl;
    cout<<"a/b의 나머지 : "<<a%b<<endl;
    return 0;
}
```

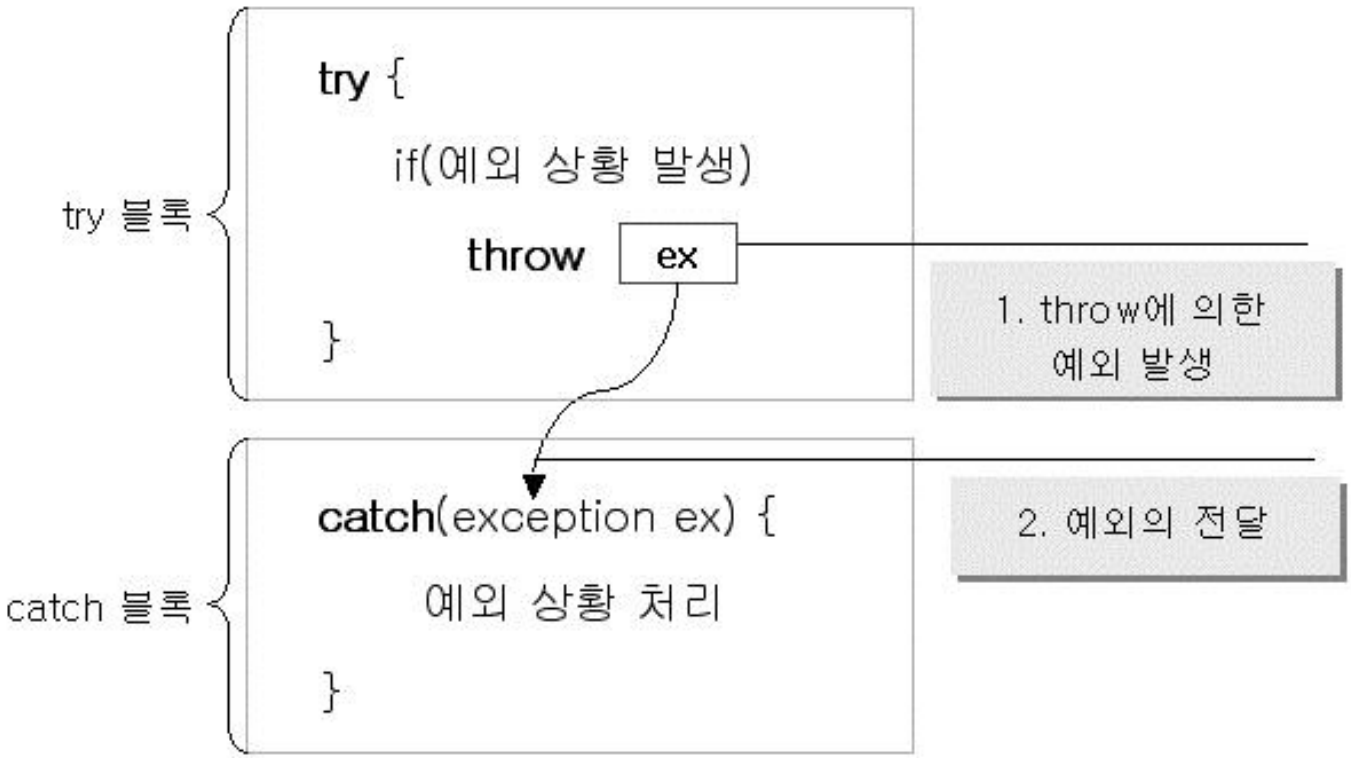
```
int main(void)
{
    int a, b;
    cout<<"두개의 숫자 입력 : ";
    cin>>a>>b;
    if(b==0){
        cout<<"입력오류!"<<endl;
    }
    else {
        cout<<"a/b의 몫 : "<<a/b<<endl;
        cout<<"a/b의 나머지 : "<<a%b<<endl;
    }
    return 0;
}
```

2. C++의 예외처리 메커니즘

- Try, catch, throw

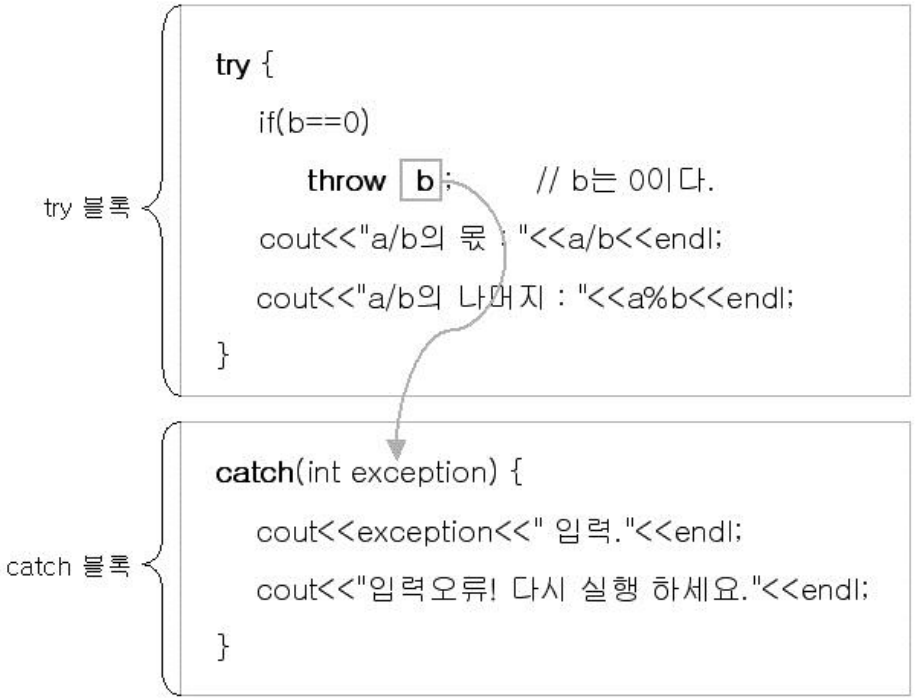
```
try {
        /* 예외 발생 예상 지역 */
}
catch(처리 되어야 할 예외의 종류) {
      /*  예외를 처리하는 코드가 존재할 위치 */
}
```

```
throw ex;   // ex를 가리켜 보통은 그냥 “예외”라고 표현을 한다.
```


try 블록
try {
if(예외 상황 발생)
throw
ex
}
1. throw에 의한
예외 발생
catch 블록
catch(exception ex) {
예외 상황 처리
}
2. 예외의 전달

- 예제

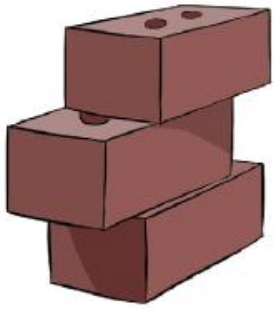
```
int main(void)
{
    int a, b;
    cout<<"두개의 숫자 입력 : ";
    cin>>a>>b;

    try{
        if(b==0)
            throw b;
        cout<<"a/b의 몫 : "<<a/b<<endl;
        cout<<"a/b의 나머지 : "<<a%b<<endl;
    }

    catch(int exception){
        cout<<exception<<" 입력."<<endl;
        cout<<"입력오류! 다시 실행 하세요."<<endl;
    }
    return 0;
}
```



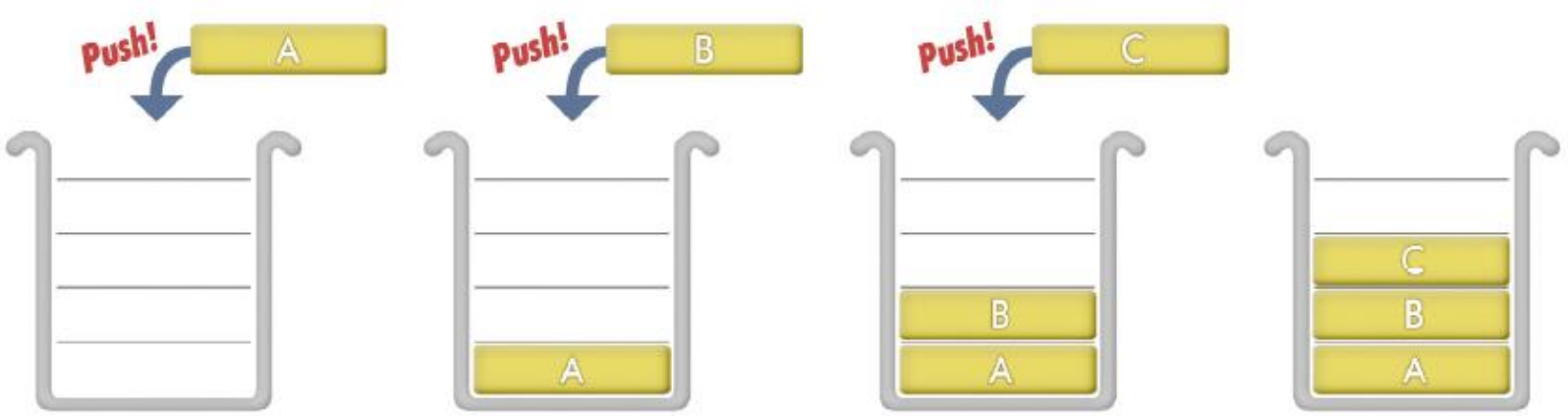
- 예외가 발생하면 try 블록의 나머지 부분 무시
- 예외 처리 후 catch 블록 이후부터 실행

3. Stack Unwinding (스택 풀기)

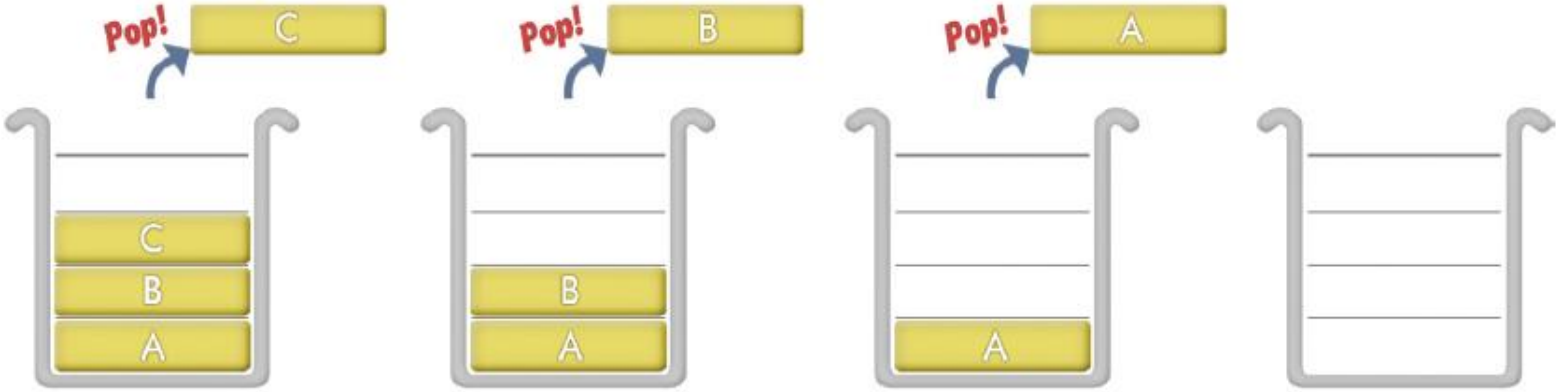
- Stack
  - 가장 나중에 들어온 자료가 가장 먼저 나가게 되는 데이터 구조
  - LIFO(Last-In, First-Out): "후입선출형 구조"
  - FILO(First-In,Last-Out): 먼저 들어온 것이 나중에 나간다



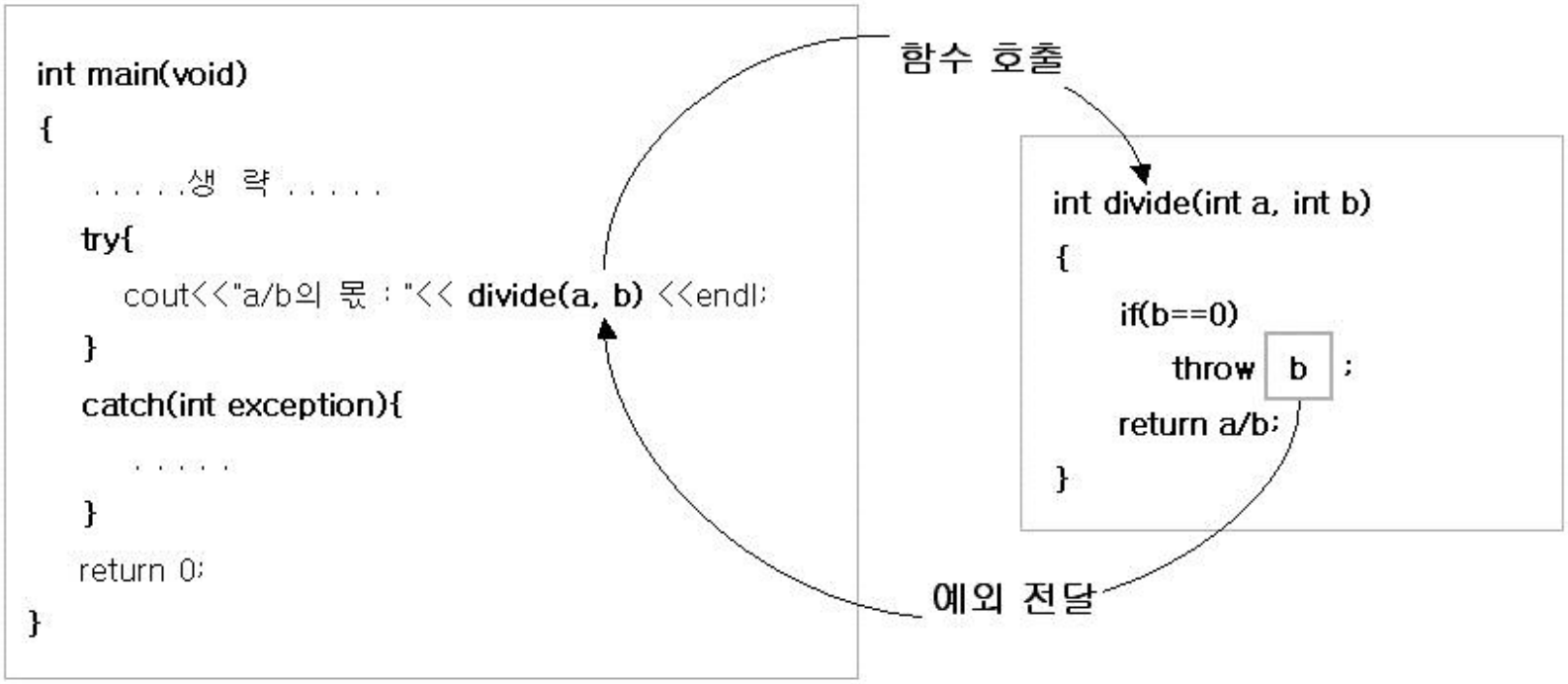
- Stack의 원리
  - Push: 스택에 새로운 자료를 추가하는 것

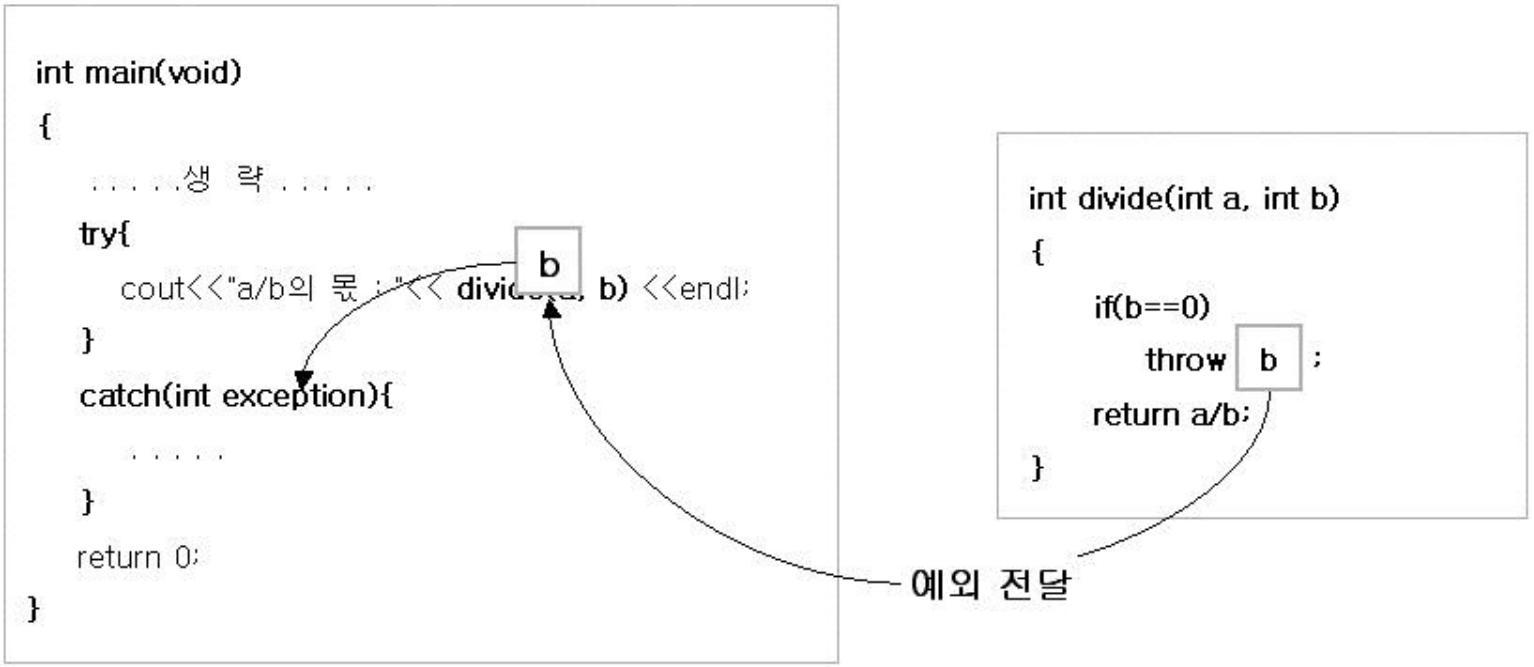


  - Pop: 스택에 저장된 값을 다시 빼내어 내는 것



- Stack 풀기

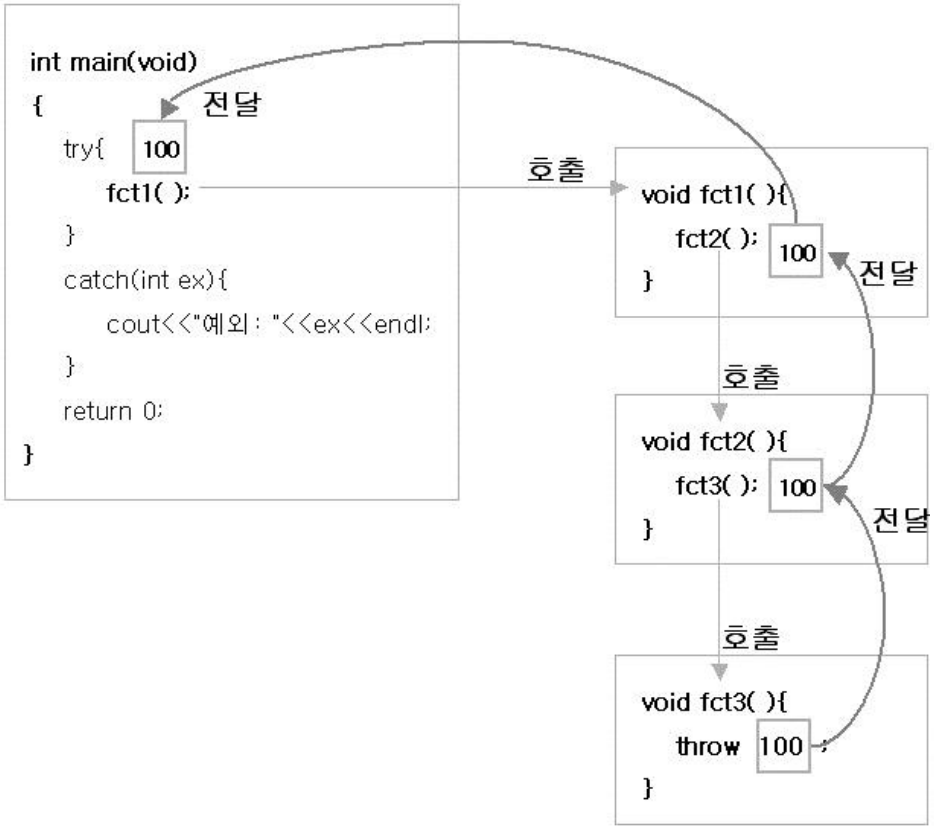

int main(void)
{
.....생 략 .....
try{
cout<<"a/b의 몫 : "<< divide(a, b) <<endl;
}
catch(int exception){
.....
}
return 0;
}
함수 호출
int divide(int a, int b)
{
if(b==0)
throw b ;
return a/b;
}
예외 전달

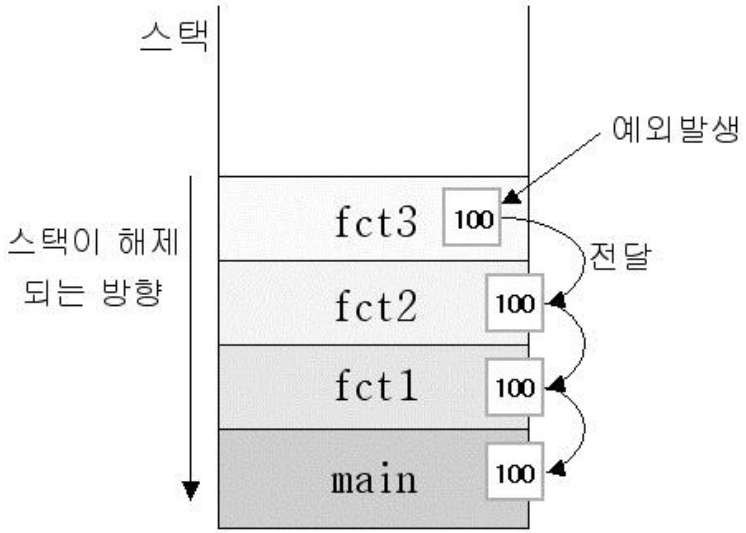

int main(void)
{
. . . . .생 략 . . . . .
try{
cout<<"a/b의 몫 :"<< divide(a, b) <<endl;
}
catch(int exception){
. . . . .
}
return 0;
}
b
int divide(int a, int b)
{
if(b==0)
throw b ;
return a/b;
}
예외 전달

- Stack 풀기 예제

```
int divide(int a, int b); // a/b의 몫만 반환

int main(void)
{
   int a, b;

   cout<<"두개의 숫자 입력 : ";
   cin>>a>>b;

   try{
        cout<<"a/b의 몫 : "<<divide(a, b)<<endl;
   }
   catch(int exception){
        cout<<exception<<" 입력."<<endl;
        cout<<"입력오류! 다시 실행 하세요."<<endl;
   }
   return 0;
}
```

```
int divide(int a, int b)
{
   if(b==0)
        throw b;
   return a/b;
}
```


int main(void)
{
try{
100
fct1( );
}
catch(int ex){
cout<<"예외 : "<<ex<<endl;
}
return 0;
}
전달
호출
void fct1( ){
fct2( );
100
}
전달
호출
void fct2( ){
fct3( );
100
}
전달
호출
void fct3( ){
throw 100 ;
}


스택
예외발생
스택이 해제
되는 방향
fct3
100
전달
fct2
100
fct1
100
main
100

- 예외가 처리되지 않으면
  - stdlib.h에 선언되어 있는 abort 함수의 호출에 의해 프로그램 종료

- 전달되는 예외 명시
  - 그 이외의 예외가 전달되면 abort 함수의 호출

```
int fct(double d) throw (int, double, char *)

{

   . . . . .

}
```

- 하나의 try, 여러 개의 catch

```
int main(void)
{
    int num;

    cout<<"input number: ";
    cin>>num;

    try{
        if(num>0)
            throw 10;  // int형 예외 전달.
        else
            throw 'm';  // char형 예외 전달.
    }
    catch(int exp){
        cout<<"int형 예외 발생"<<endl;
    }
    catch(char exp){
        cout<<"char형 예외 발생"<<endl;
    }
    return 0;
}
```

4. 예외상황을 나타내는 클래스의 설계

- 예외 상황을 나타내는 클래스 & 객체
  - 예외 클래스, 예외 객체
  - 일반 클래스, 객체와 다르지 않다.
  - 예외 상황을 알려주기 위한 용도로 사용
- 예외는 클래스의 객체로 표현되는 것이 일반적

5. 예외처리에 대한 나머지 문법 요소

- 모든 예외 처리 catch 블록

```
try{
    . . . . .
 }
 catch(...) {  // ... 선언은 모든 예외를 다 처리 하겠다는 선언
    . . . . .
 }
```

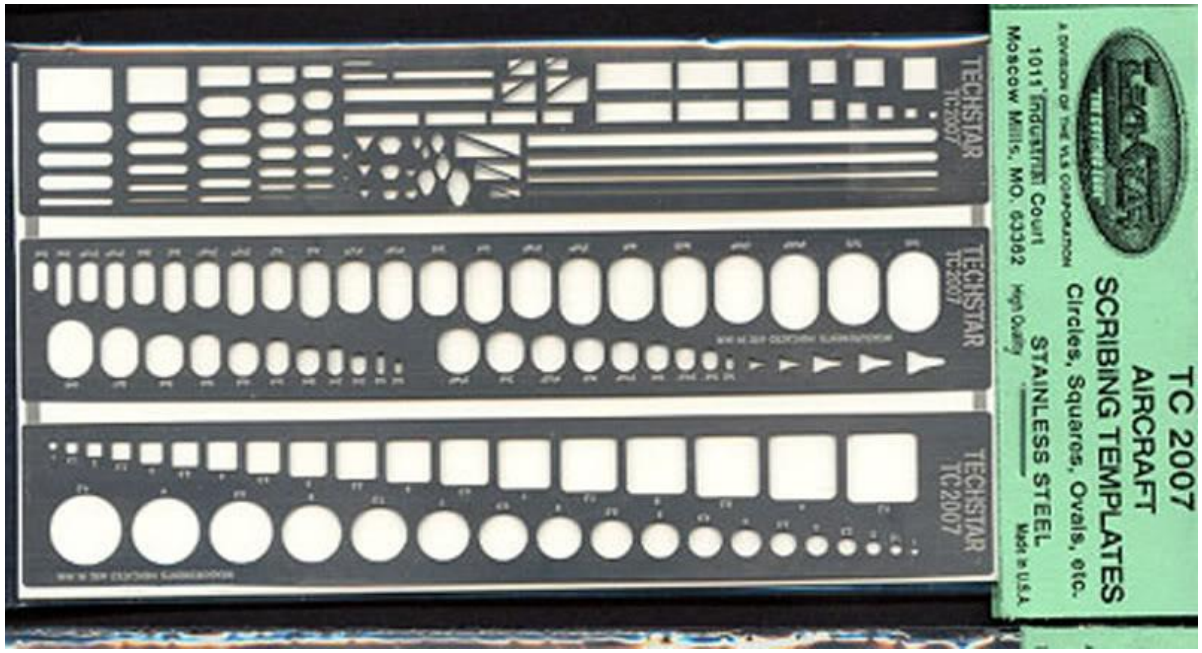
- 예외 다시 던지기

```
try{
    . . . . .
 }
 catch(Exception& t) {
    . . . . .
   throw;
 }
```

11-2. 템플릿

1.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;
}
```

일반화 프로그래밍

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

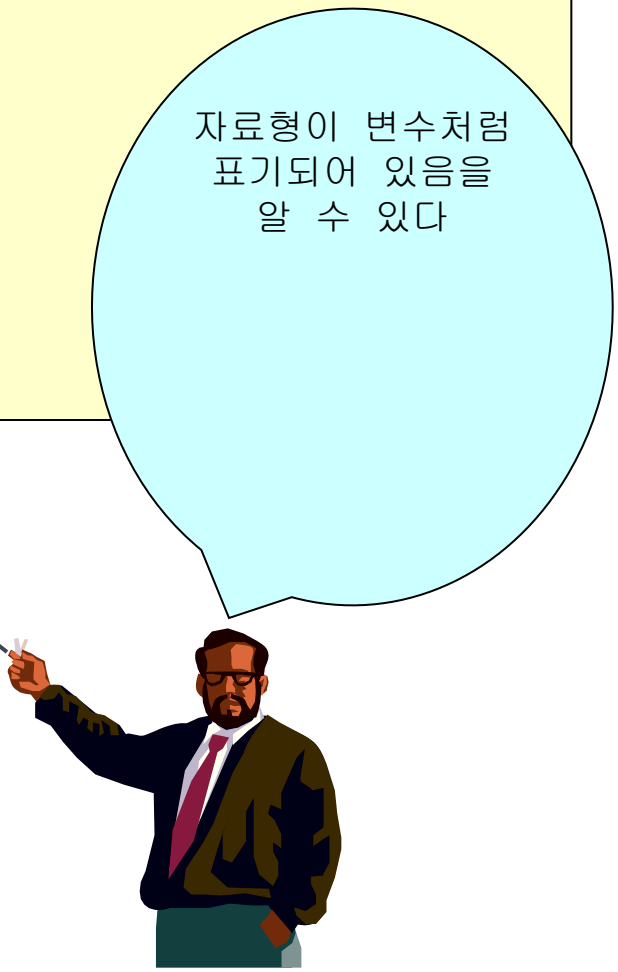
템플릿 함수

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

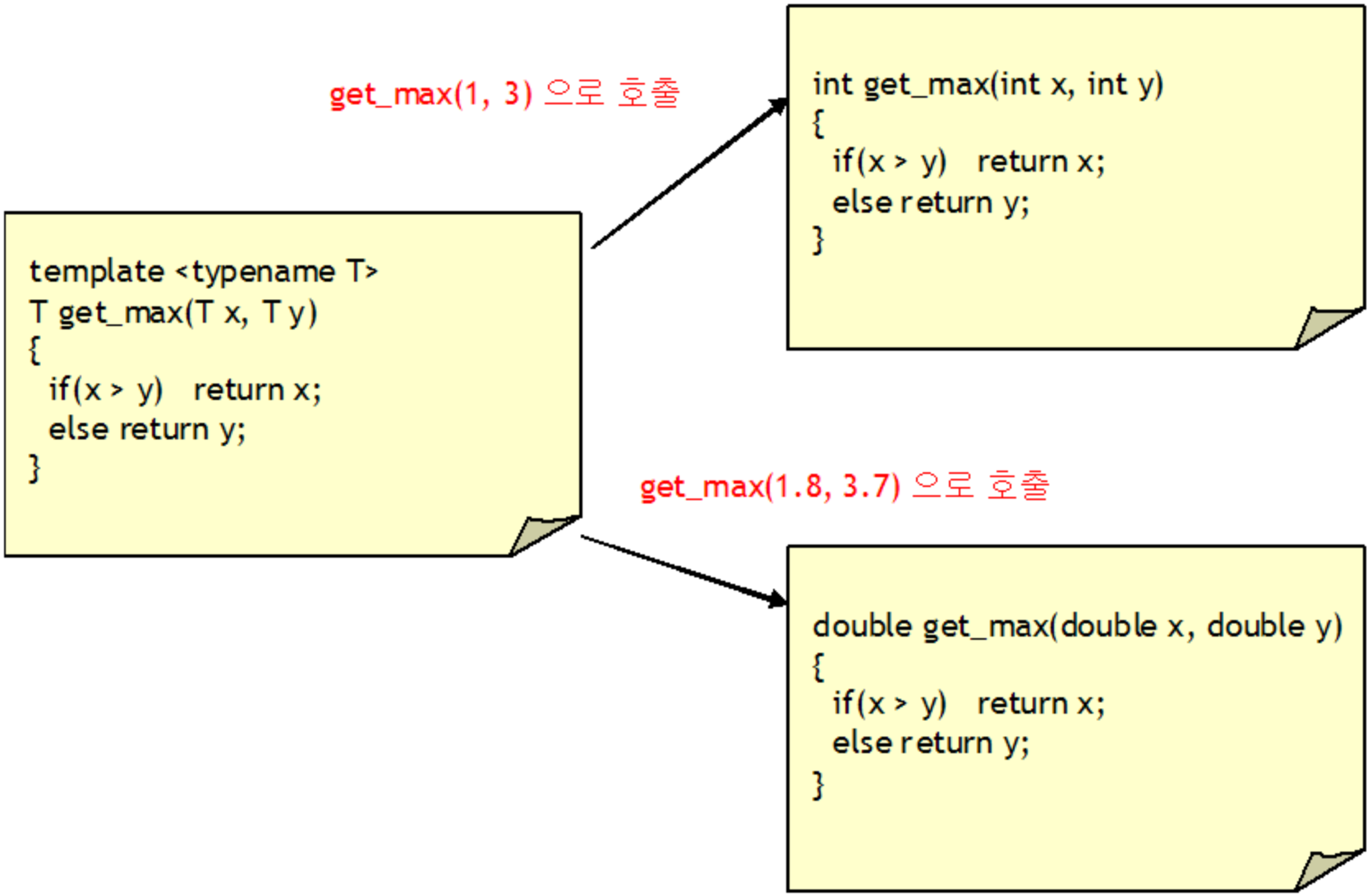
```
int get_max(int  x , int y)
{
  if( x > y)   return x;
  else return y;
}
```

get_max()

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



템플릿 함수의 사용


get_max(1, 3) 으로 호출
int get_max(int x, int y)
{
if(x > y) return x;
else return y;
}
template <typename T>
T get_max(T x, T y)
{
if(x > y) return x;
else return y;
}
get_max(1.8, 3.7) 으로 호출
double get_max(double x, double y)
{
if(x > y) return x;
else return y;
}

예제

get_max.cpp

```
#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()
{
  // 아래의 문장은 정수 버전 get_max()를 호출한다.
  cout << get_max(1, 3) << endl;

  // 아래의 문장은 실수 버전 get_max()를 호출한다.
  cout << get_max(1.2, 3.9) << endl;

  return 0;
}
```

실행 결과

```
3
3.9
계속하려면 아무 키나 누르십시오 . . .
```

2. 템플릿 함수의 특수화

```
template <typename T>                    // 함수 템플릿으로 정의
void print_array(T[] a, int n)
{
    for(int i=0;i<n; i++)
            cout << a[i] << " ";
    cout << endl;
}

template <>                              // 템플릿 특수화
void print_array(char[] a, int n)     // 매개 변수가 char인 경우에는 이 함수가
    호출된다.
{
    cout << a << endl;
}
```

함수 템플릿과 함수 중복

swap_values.cpp

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

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

템플릿함수

함수 중복

```
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;
}
```

실행 결과

```
200 100
This is a second string This is a first string
계속하려면 아무 키나 누르십시오 . . .
```

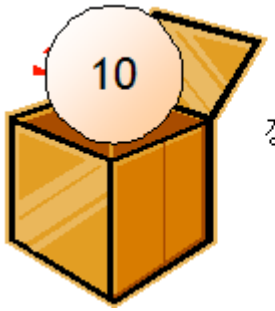
```
template<typename T1, typename T2>
void copy(T1 a1[], T2 a2[], int n)
{
    for (int i = 0; i < n; ++i)
        a1[i] = a2[i];
}
```

2.클래스 템플릿

- 클래스 템플릿(class template): 클래스를 찍어내는 틀(template)

```
template <typename 타입이름, ...> class 클래스이름
{
...

}
```

- 예제: 하나의 값을 저장하고 있는 박스



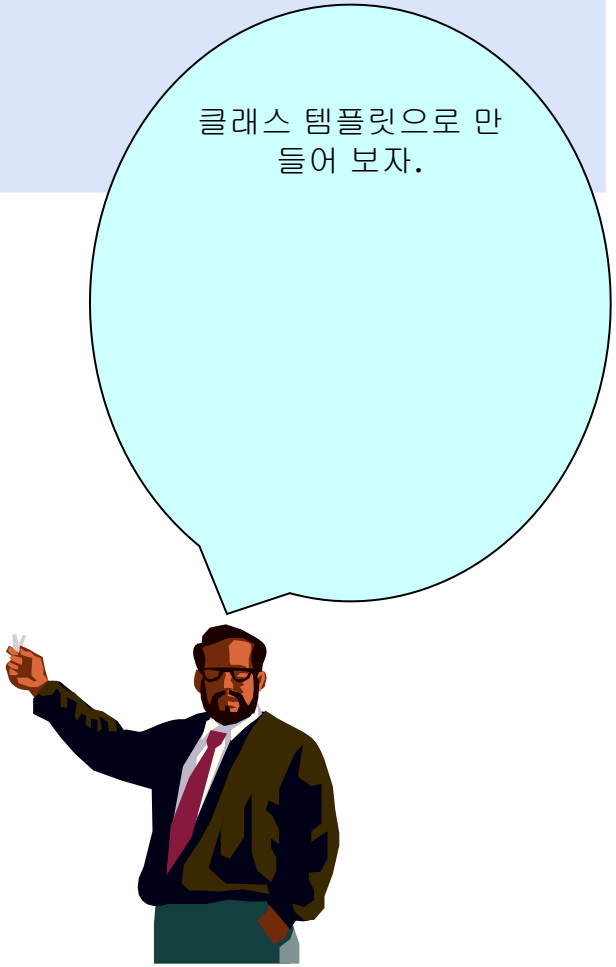
정수를 저장하는 상자

예제

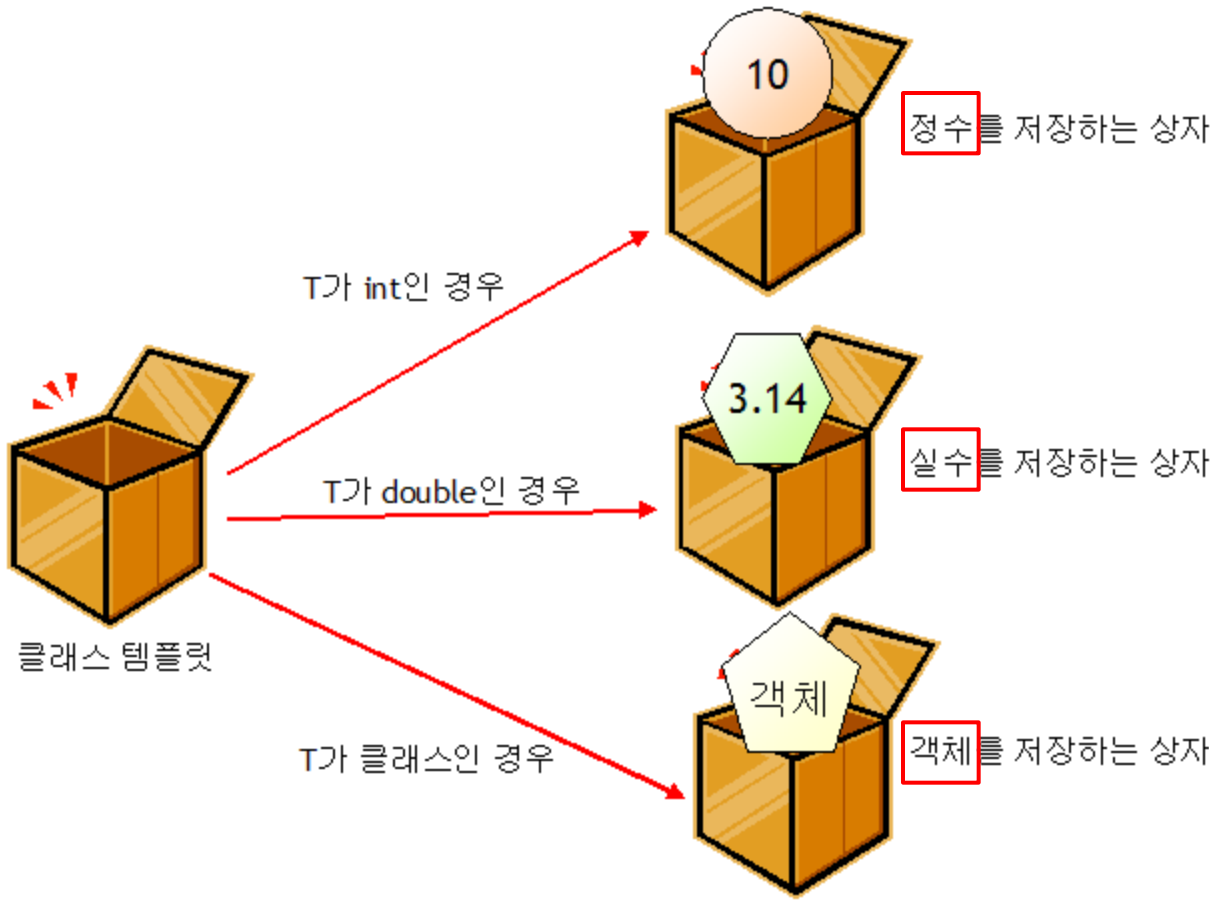
```
class Box {
    int data;
public:
    Box() {  }
    void set(int value) {
        data = value;
    }
    int get() {
        return data;
    }
};

int main()
{
   Box box;
   box.set(100);
   cout << box.get() << endl;
   return 0;
}
```

실행 결과

```
100
계속하려면 아무 키나 누르십시오 . . .
```



클래스 템플릿 버전


10
정수를 저장하는 상자
T가 int인 경우
3.14
실수를 저장하는 상자
T가 double인 경우
클래스 템플릿
객체
객체를 저장하는 상자
T가 클래스인 경우

클래스 템플릿 정의

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

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;
    }
};
```

클래스 템플릿

```
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;
}
```

정수 버전 클래스

실수 버전 클래스

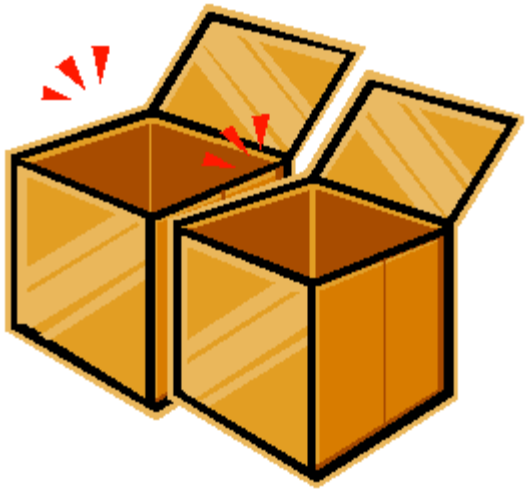
실행 결과

```
100
3.14159
계속하려면 아무 키나 누르십시오 . . .
```

```
template <typename T>
class Box {
    T data; // T는 타입(type)을 나타낸다.
public:
    Box();
    void set(T value);
    T get();
};

template <typename T>
Box<T>::Box() {
}

template <typename T>
void Box<T>::set(T value) {
  data = value;
}

template <typename T>
T Box<T>::get() {
  return data;
}
```

두개의 타입 매개 변수

- 두 개의 데이터를 저장하는 클래스 Box2



Box2 클래스 템플릿

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;
   }
};
```

두개의 타입 매개변수를 가지는 클래스 템플릿

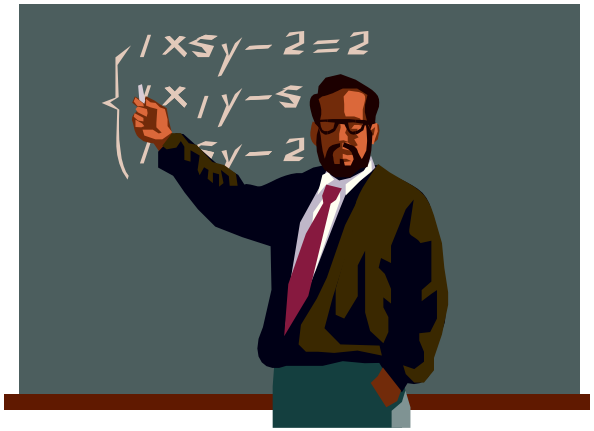
예제

```
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;
}
```

실행 결과

```
(10, 3.14)
계속하려면 아무 키나 누르십시오 . . .
```

Q & A



참고문헌

- 뇌를 자극하는 C++ 프로그래밍, 이현창, 한빛미디어1
- 열혈 C++ 프로그래밍(개정판), 윤성우, 오렌지미디어
- C++ ESPRESSO, 천인국 저, 인피니티북스
- 명품 C++ Programming, 황기태 , 생능출판사

추가 자료

예외 객체의 사용

- 객체를 예외로 던지면 다양한 정보를 함께 전달할 수 있다.

```
class MyException
{
public:
          const void* sender;           // 예외를 던진 객체의 주소
          const char* description;      // 예외에 대한 설명
          int info;                     // 부가 정보

          MyException(const void* sender, const char* description, int info)
          {
                    this->sender = sender;
                    this->description = description;
                    this->info = info;
          }
};
```

```
// 원소의 값을 바꾼다
void DynamicArray::SetAt(int index, int value)
{
          // 인덱스의 범위가 맞지 않으면 예외를 던진다
          if (index < 0 || index >= GetSize())
                    throw MyException( this, "Out of Range!!!", index);

          arr[index] = value;
}
```

- 다형성을 사용해서 일관된 관리를 할 수 있다.

```
// 인덱스와 관련된 예외
class OutOfRangeException : public MyException
{
        ...
};

// 메모리와 관련된 예외
class MemoryException : public  MyException
{
        ...
};
```

```
        try
        {
                // OutOfRangeException& 혹은 MemoryException& 타입의
                // 예외 객체를 던진다고 가정
        }
        catch(MyException& ex)
        {
                // OutOfRangeException 과 MemoryException 모두
                // 여기서 잡을 수 있다.
                cout << "예외에 대한 설명= " << ex.description << "\n";
        }
```

구조적 예외 처리의 규칙

- 예외는 함수를 여러 개 건너서도 전달할 수 있다.

```
int main()
{
	try
	{
		A();
	}
	catch(int ex)
	{
		cout << "예외 = " << ex << "\n";
	}

	return 0;
}

void A()
{
	B();
}

void B()
{
	C();
}

void C()
{
	throw 337;
}
```

- 받은 예외를 다시 던질 수 있다.

```
int main()
{
        try {
                A();
        }
        catch(char c){
                cout << "main() 함수에서 잡은 예외 = " << c << "\n";
        }

        return 0;
}

void A()
{
        try {
                B();
        }
        catch(char c) {
                cout << "A() 함수에서 잡은 예외 = " << c << "\n";

                throw;
        }
}

void B()
{
        throw 'X';
}
```

구조적 예외 처리의 규칙(3)

- try 블럭 하나에 여러 개의 catch 블럭이 이어질 수 있다.

```
int main()
{
	try
	{
		A();
	}
	catch(MemoryException& ex)
	{
		cout << "MemoryException 타입으로 잡혔음\n";
	}
	catch(OutOfRangeException& ex)
	{
		cout << "OutOfRangeException 타입으로 잡혔음\n";
	}
	catch(...)
	{
		cout << "그 밖의 타입\n";
	}

	return 0;
}

void A()
{
	// throw 100;
	throw OutOfRangeException(NULL, 0);
}
```

구조적 예외 처리의 규칙(4)

- 실행 결과

- Catch 블록이 여럿인 경우

```
int main()
{
    try
    {
        A();
    }
    catch(MemoryException& ex)
    {
        // 생략
    }
    catch(OutOfRangeException& ex)
    {
        // 생략
    }
    catch(...)
    {
        // 생략
    }

    return 0;
}
```

```
void A()
{
    throw OutOfRangeException(NULL, 0);
}
```

① MemoryException& ex = OutOfRangeException(Null, 0)
위와 같은 가상의 코드는 합법적이지 못하다
그래서 이 catch 블록은 예외를 잡을 수 없다

② OutOfRangeException& ex = OutOfRangeException(Null, 0)
위와 같은 가상의 코드가 합법적이다
그래서 이 catch 블록이 실행된다

③ 앞의 catch 블록이 실행되었기 때문에
이 catch 블록은 실행되지 않는다

구조적 예외 처리의 규칙(5)

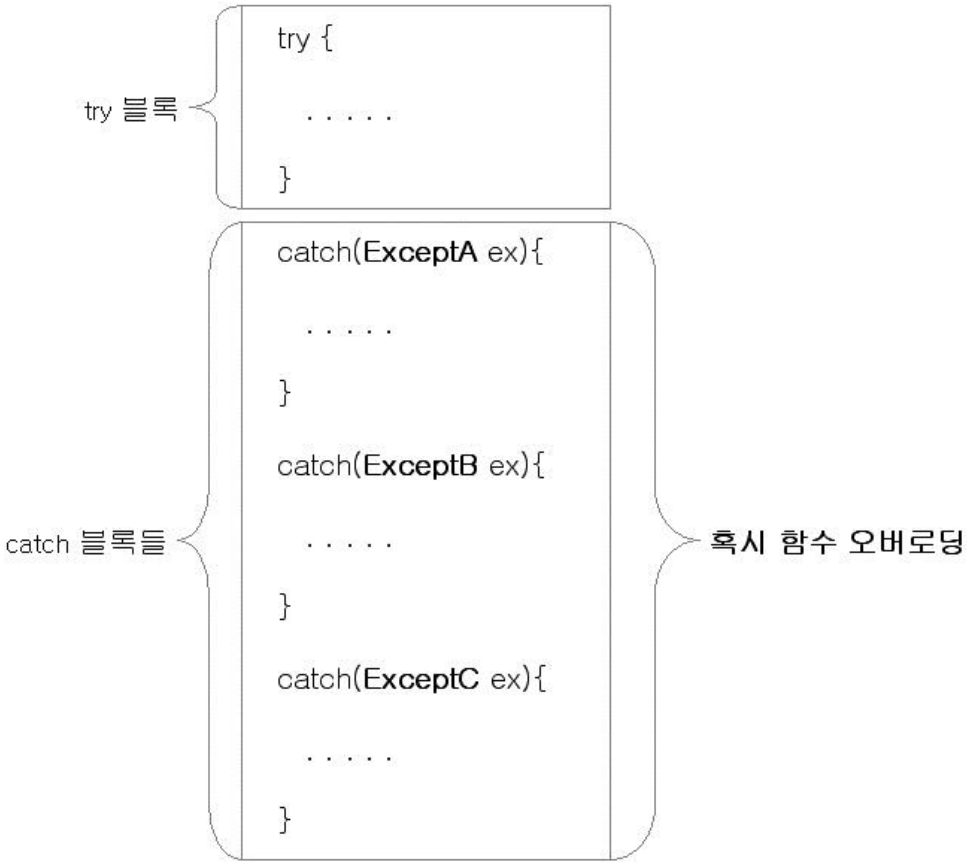
- 예외 객체는 레퍼런스로 받는 것이 좋다.

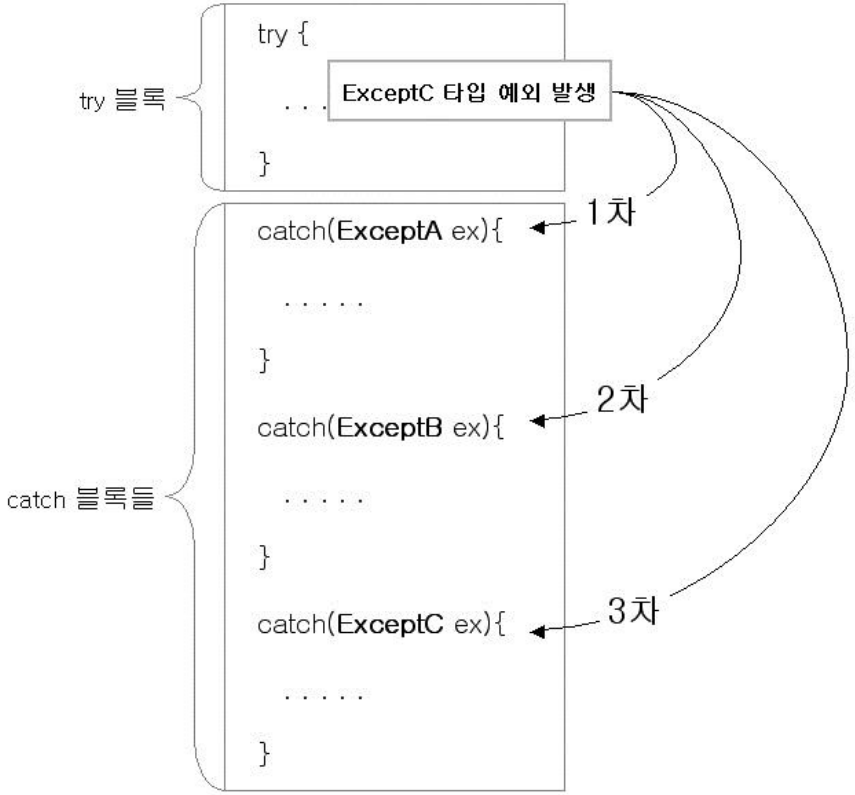
```
try
{
          MyException e( this, “객체”, 0 );
          throw e;
}
catch( MyException& ex)
{
          cout << ex.description << “\n”;
}
```

- 레퍼런스나 포인터가 아닌 객체의 타입으로 받는 경우에는 불필요한 복사가 발생하므로 비효율적이다.
- 객체를 동적으로 할당해서 던지고 포인터 타입으로 받는 경우에 불필요한 복사가 발생하지는 않지만, 메모리 할당과 해제를 신경 써야 하므로 불편하다.

1. 예외를 나타내는 클래스와 상속

- catch 블록에 예외가 전달되는 방식





2. new 연산자에 의해 전달되는 예외

- new 연산자가 메모리 공간 할당을 실패 했을 때 발생 시키는 예외

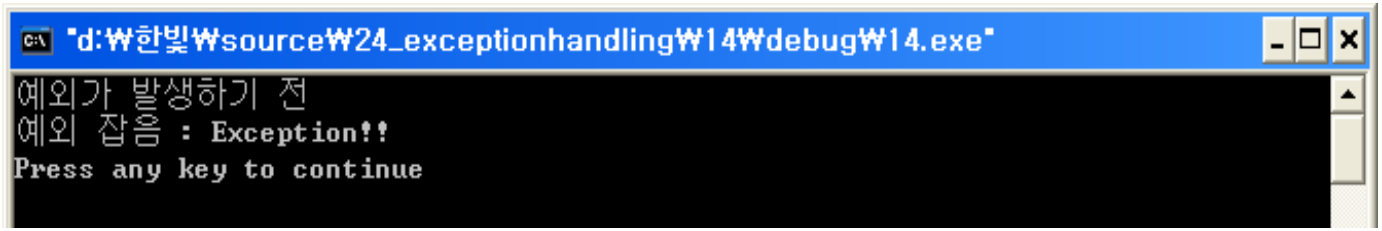
```
using std::bad_alloc;

int main(void)
{
    try{
        int i=0;
        while(1){
            cout<<i++<<"번째 할당"<<endl;
            double(*arr)[10000]=new double[10000][10000];
        }
    }
    catch(bad_alloc ex){
        ex.what();
        cout<<endl<<"END"<<endl;
    }

    return 0;
}
```

4. 리소스의 정리

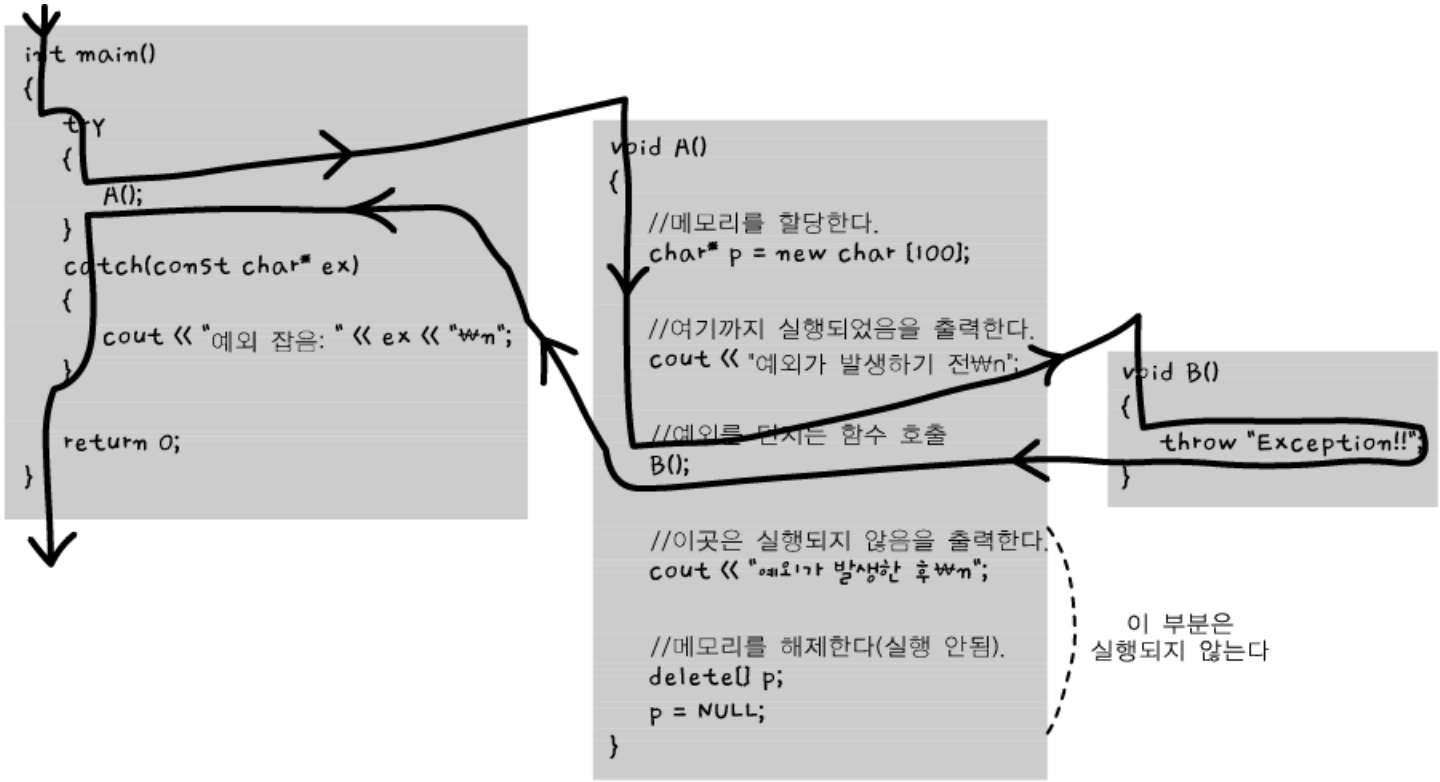
- 리소스를 정리하기 전에 예외가 발생한 경우의 문제점 확인

```
try
{
        // 메모리를 할당한다.
        char* p = new char [100];

        // 여기까지 실행되었음을 출력한다.
        cout << "예외가 발생하기 전\n";

        // 예외를 던진다.
        throw "Exception!!";

        // 이곳은 실행되지 않음을 출력한다.
        cout << "예외가 발생한 후\n";

        // 메모리를 해제한다. (실행 안됨)
        delete[] p;
        p = NULL;
}
catch(const char* ex)
{
        cout << "예외 잡음 : " << ex << "\n";
}
```

- 실행 결과


"d:₩한빛₩source₩24_exceptionhandling₩14₩debug₩14.exe"
예외가 발생하기 전
예외 잡음 : Exception!!
Press any key to continue

- 실행의 흐름


int main()
{
try
{
A();
}
catch(const char* ex)
{
cout << "예외 잡음: " << ex << "₩n";
}
return 0;
}
void A()
{
//메모리를 할당한다.
char* p = new char [100];
//여기까지 실행되었음을 출력한다.
cout << "예외가 발생하기 전₩n";
//예외를 던지는 함수 호출
B();
//이곳은 실행되지 않음을 출력한다.
cout << "예외가 발생한 후₩n";
//메모리를 해제한다(실행 안됨).
delete[] p;
p = NULL;
}
void B()
{
throw "Exception!!";
}
이 부분은
실행되지 않는다

5. 소멸자를 이용한 리소스의 정리

- 예외가 발생한 경우라도 객체의 소멸자는 반드시 호출되므로 소멸자를 사용해서 리소스 릭을 방지할 수 있다.

```
// 스마트 포인터 클래스
class SmartPointer
{
public:
          SmartPointer(char* p)
                    : ptr(p)
          {
          }
          ~SmartPointer()
          {
                    // 소멸자가 호출되는 것을 확인한다
                    cout << "메모리가 해제된다!!\n";

                    delete[] ptr;
          }

public:
          char * const ptr;
};
```

- 소멸자를 사용해서 리소스 릭을 방지하는 예

```
try
{
        // 메모리를 할당한다.
        char* p = new char [100];

        // 할당된 메모리의 주소를 스마트 포인터에 보관한다.
        SmartPointer sp(p)

        // 여기까지 실행되었음을 출력한다.
        cout << "예외가 발생하기 전\n";

        // 예외를 던진다.
        throw "Exception!!";

        // 이곳은 실행되지 않음을 출력한다.
        cout << "예외가 발생한 후\n";

        // 메모리를 해제해줄 필요가 없다.
        // delete[] p;
        // p = NULL;
}
catch(const char* ex)
{
        cout << "예외 잡음 : " << ex << "\n";
}
```

6. 생성자에서의 예외처리

- 생성자에서 예외가 발생한 경우에는 객체가 생성되지 않은 것으로 간주되므로 소멸자가 호출되지 않는다.
- 그러므로, 생성자에서 예외가 발생한 경우에는 반드시 생성자 안쪽에서 리소스를 정리해야 한다.

```
DynamicArray::DynamicArray(int arraySize)
{
        try
        {
                // 동적으로 메모리를 할당한다.
                arr = new int [arraySize];

                // 배열의 길이 보관
                size = arraySize;

                // 여기서 고의로 예외를 발생시킨다.
                throw MemoryException( this, 0);
        }
        catch(...)
        {
                cout << "여기는 실행된다!!\n";

                delete[] arr;       // 리소스를 정리한다.

                throw;              // 예외는 그대로 던진다.
        }
}
```

7. 소멸자에서의 예외처리

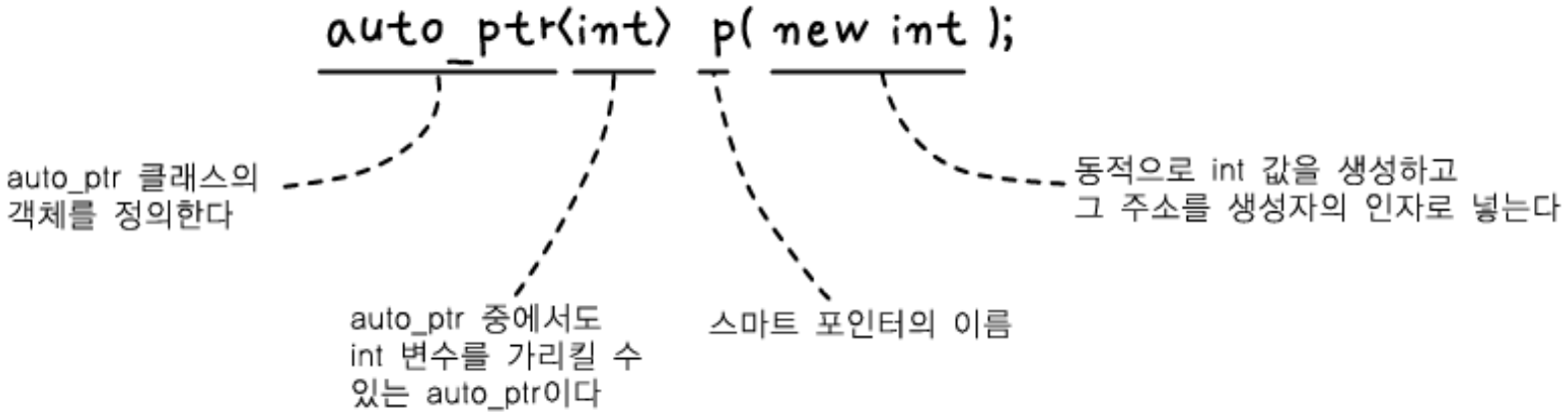
- 소멸자에서 예외가 발생하는 경우에는 프로그램이 비정상 종료할 수 있으므로, 소멸자 밖으로 예외가 던져지지 않게 막아야 한다.

```
DynamicArray::~DynamicArray()
{
        try
        {
                // 메모리를 해제한다.
                delete[] arr;
                arr = 0;
        }
        catch(...)
        {
        }
}
```

auto_ptr

- C++에서 기본적으로 제공하는 스마트 포인터 클래스

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

int main()
{
        // 스마트 포인터 생성
        auto_ptr<int> p( new int );

        // 스마트 포인터의 사용
        *p = 100;

        // 메모리를 따로 해제해 줄 필요가 없다

        return 0;
}
```



bad_alloc

- 동적 메모리 할당 실패시 던져지는 예외 클래스

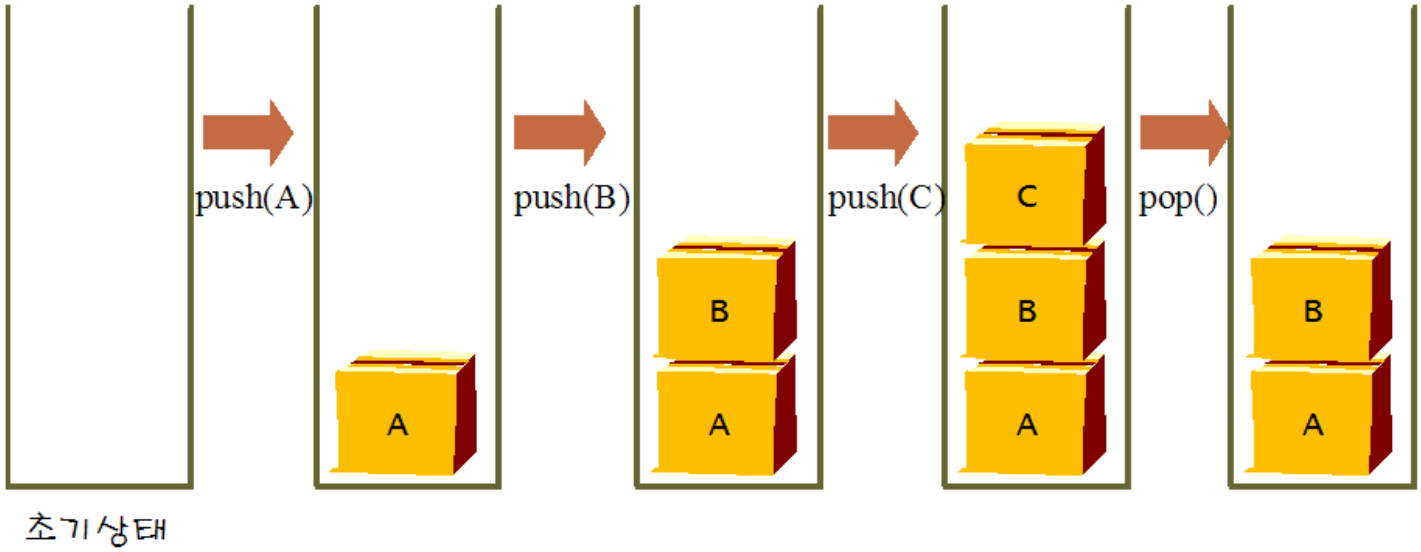
```
#include <new>
#include <iostream>
using namespace std;

int main()
{
        try
        {
                // 많은 양의 메모리를 할당한다.
                char* p = new char [0xfffffff0];

        }
        catch(bad_alloc& ex)
        {
                cout << ex.what() << "\n";
        }
        return 0;
}
```

- 실행 결과

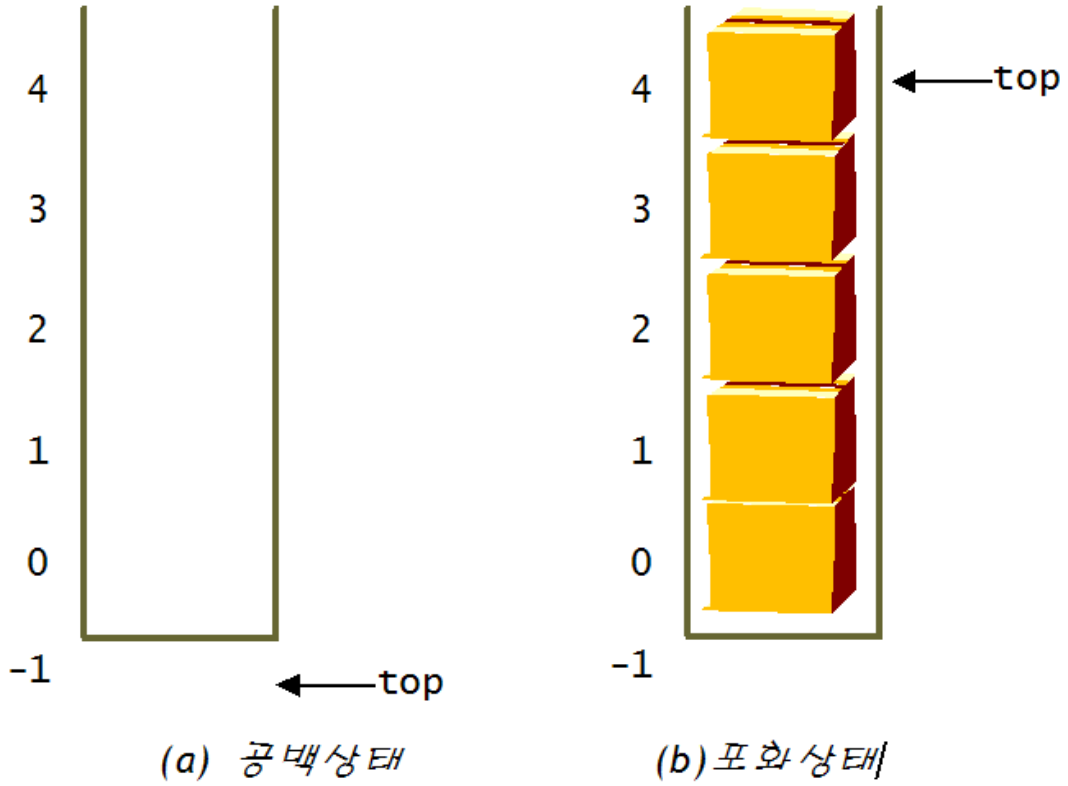
추가 예제) 템플릿 예제 - 스택

스택의 연산들

- is_empty(s) ::= 스택이 비어있는지를 검사한다.
- is_full(s) ::= 스택이 가득 찼는가를 검사한다.
- push(s, e) ::= 스택의 맨 위에 요소 e를 추가한다.
- pop(s) ::= 스택의 맨 위에 있는 요소를 삭제한다.



스택의 공백 상태와 포화 상태



(a) 공백상태 (b) 포화상태

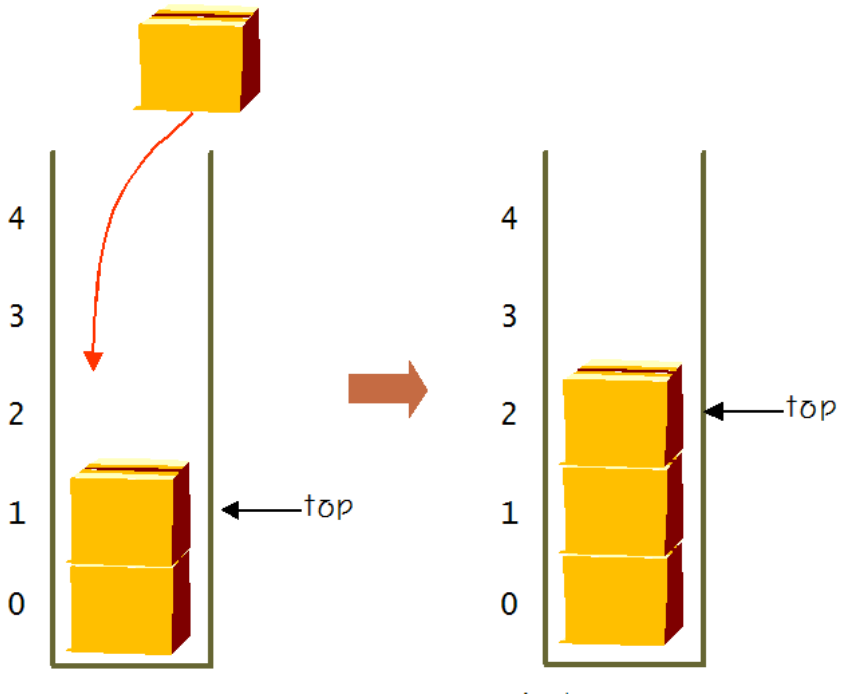
isEmpty() , isFull()

```
isEmpty()

if top = -1
    then return TRUE
    else return FALSE
```

```
isFull()

if top = (MAX_STACK_SIZE-1)
    then return TRUE
    else return FALSE
```

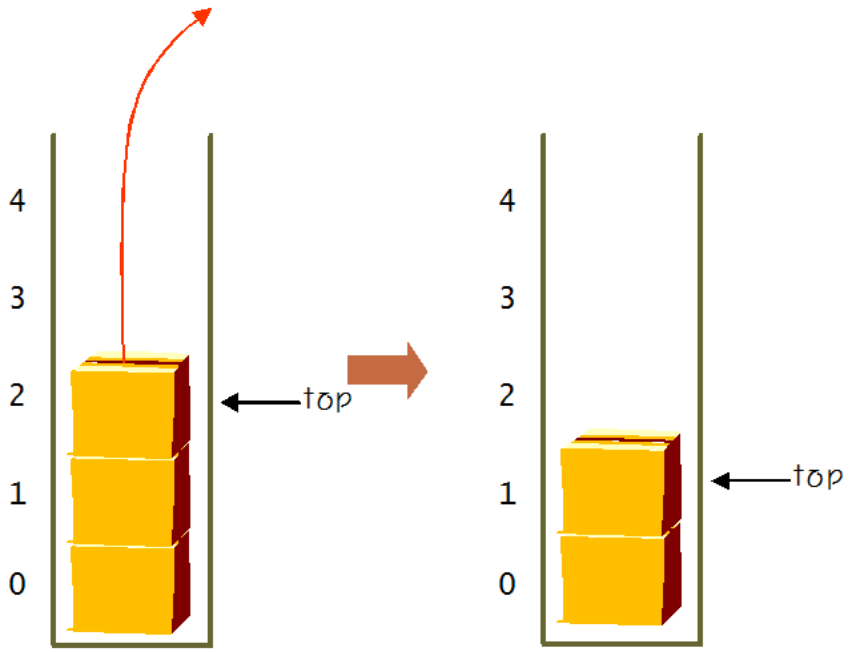
push()

```
push(x)

if isFull()
    then error "overflow"
    else top←top+1
         stack[top]←x
```



pop()

```
pop(x)

if isEmpty()
    then error "underflow"
    else e←stack[top]
        top←top-1
        return e
```



스택의 구현

```
#include <iostream>
using namespace std;
// 예외 처리를 위한 클래스
class FullStack
{} ;

// 예외 처리를 위한 클래스
class EmptyStack
{};
```

```
template <class T>
class Stack {
private:
  T*  s;
  int    size;
  int    top;
public:
  Stack(int n = 100) : size(n), top(-1)
  {
                s = new T[size];
     }
  ~Stack() { delete []s; }
  void  push(T v);
  T pop();
  bool  isEmpty() const { return top == -1;}
  bool  isFull() const { return top == size - 1;}
};
```

스택의 구현

```
template< typename T >
void Stack< T >::push( T v )
{
  if ( isFull() )
            throw FullStack();
  s[ ++top ] = v;
}

template< typename T >
T Stack< T >::pop( )
{
  if ( isEmpty() )
            throw EmptyStack();
  return s[ top-- ];
}
```

```
int main()
{
    Stack<int> s;  // 크기가 100인 정수형 스택
    s.push(100);
    s.push(200);
    s.push(300);
    s.push(400);
    cout << s.pop() << endl;
    cout << s.pop() << endl;
    cout << s.pop() << endl;
    cout << s.pop() << endl;
    return 0;
}
```

```
400
300
200
100
계속하려면 아무 키나 누르십시오 . . .
```

stack2.cpp

```
...     // 앞의 스택 클래스 포함
int main()
{
   Stack<char> s;  // 크기가 100인 문자형 스택
   string str = "madamimadam";

   for(int i=0;i<str.length(); i++)
        s.push(str[i]);

   for(int i=0;i<str.length(); i++) {
        if( s.pop() != str[i] ) {
                cout << "주어진 문자열은 회문이 아님" << endl;
                return 0;
        }
   }
   cout << "주어진 문자열은 회문임" << endl;
   return 0;
}
```

실행 결과

```
주어진 문자열은 회문임
계속하려면 아무 키나 누르십시오 . . .
```