예외처리
(Exception Handling)

박 종 혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

예외상황과 예외처리의 이해

예외상황을 처리하지 않았을 때의 결과

√ 예외상황은 프로그램 실행 중에 발생하는 문제의 상황을 의미한다.

√ 예외상황의 예

나이를 입력하라고 했는데, 0보다 작은 값이 입력됨.

나눗셈을 위해서 두 개의 숫자를 입력 받았는데, 제수로 0이 입력됨.

주민등록번호 13자리만 입력하라고 했는데 중간에 –가 삽입됨.

√ 이렇듯 예외는 문법적 오류가 아닌, 프로그램 논리에 맞지 않는 오류를 뜻한다.

```
int main(void)
{
    int num1, num2;
    cout<<"두 개의 숫자 입력: ";
    cin>>num1>>num2;

    cout<<"나눗셈의 몫: "<< num1/num2 <<endl;
    cout<<"나눗셈의 나머지: "<< num1%num2 <<endl;
    return 0;
}
```

실행결과1

```
두 개의 숫자 입력: 9 2
나눗셈의 몫: 4
나눗셈의 나머지: 1
```

실행결과2

```
두 개의 숫자 입력: 7 0
     <더 이상 실행되지 않고 프로그램이 중단됩니다>
```

if문을 이용한 예외의 처리

√ if문을 이용해서 예외를 발견하고 처리하면, 예외처리 부분과 일반적인 프로그램의 흐름을 쉽게 구분할 수 없다.

√ if문은 일반적은 프로그램의 논리를 구현하는데 주로 사용된다.

√ 그래서 C++은 별도의 예외처리 메커니즘을 제공하고 있다.

if문을 통해서 예외를 발견

if문 안에서 예외를 처리!

```
int main(void)
{
    int num1, num2;
    cout<<"두 개의 숫자 입력: ";
    cin>>num1>>num2;
    if(num2==0)
    {
        cout<<"제수는 0이 될 수 없습니다."<<endl;
        cout<<"프로그램을 다시 실행하세요."<<endl;
    }
    else
    {
        cout<<"나눗셈의 몫: "<< num1/num2 <<endl;
        cout<<"나눗셈의 나머지: "<< num1%num2 <<endl;
    }
    return 0;
}
```

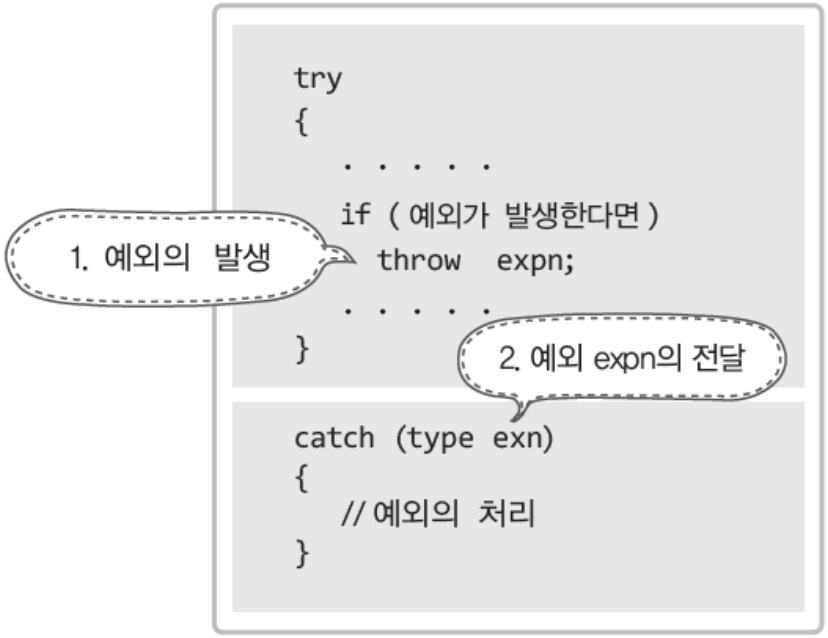
num2에 0이 입력됨으로 인해서 예외가 발생

```
두 개의 숫자 입력: 7 0
제수는 0이 될 수 없습니다.
프로그램을 다시 실행하세요.
```

실행결과

C++의 예외처리 메커니즘

C++의 예외처리 메커니즘 이해: try, catch, throw



√ try 블록은 예외발생에 대한 검사범위를 지정하는데 사용된다.

√ catch 블록은 try 블록에서 발생한 예외를 처리하는 영역으로 그 형태가 마치 반환형 없는 함수와 같다.

√ throw는 예외의 발생을 알리는 역할을 한다.

√ try~catch는 하나의 문장이므로 그 사이에 다른 문장이 삽입될 수 없다.

예외처리 메커니즘의 적용

```
int main(void)
{
    int num1, num2;
    cout<<"두 개의 숫자 입력: ";
    cin>>num1>>num2;
    try
    {
        if(num2==0)
            throw num2;
        cout<<"나눗셈의 몫: "<< num1/num2 <<endl;
        cout<<"나눗셈의 나머지: "<< num1%num2 <<endl;
    }
    catch(int expn)
    {
        cout<<"제수는 "<<expn<<"이 될 수 없습니다."<<endl;
        cout<<"프로그램을 다시 실행하세요."<<endl;
    }
    cout<<"end of main"<<endl;
    return 0;
}
```

실행결과1

```
두 개의 숫자 입력: 9 2
나눗셈의 몫: 4
나눗셈의 나머지: 1
end of main
```

실행결과2

```
두 개의 숫자 입력: 7 0
제수는 0이 될 수 없습니다.
프로그램을 다시 실행하세요.
end of main
```

예외의 발생으로 인해서 try 블록 내에서 throw절이 실행이 되면, try 블록의 나머지 부분은 실행이 되지 않는다!

실행의 흐름을 이해하는 것이 중요!

실행의 흐름은 try 블록 안으로 들어간다. 그 안에서 예외가 발생하면 이후에 등장하는 catch 블록을 실행하게 된다.

예외가 발생하지 않으면 try 블록을 빠져 나와 try~ catch 블록 이후를 실행하게 된다.

try 블록을 묶는 기준

잘 묶인 예!

```
try
{
    if(num2==0)
        throw num2;
    cout<<"나눗셈의 몫: "<< num1/num2 <<endl;
    cout<<"나눗셈의 나머지: "<< num1%num2 <<endl;
}
catch(int expn) { . . . . }
```

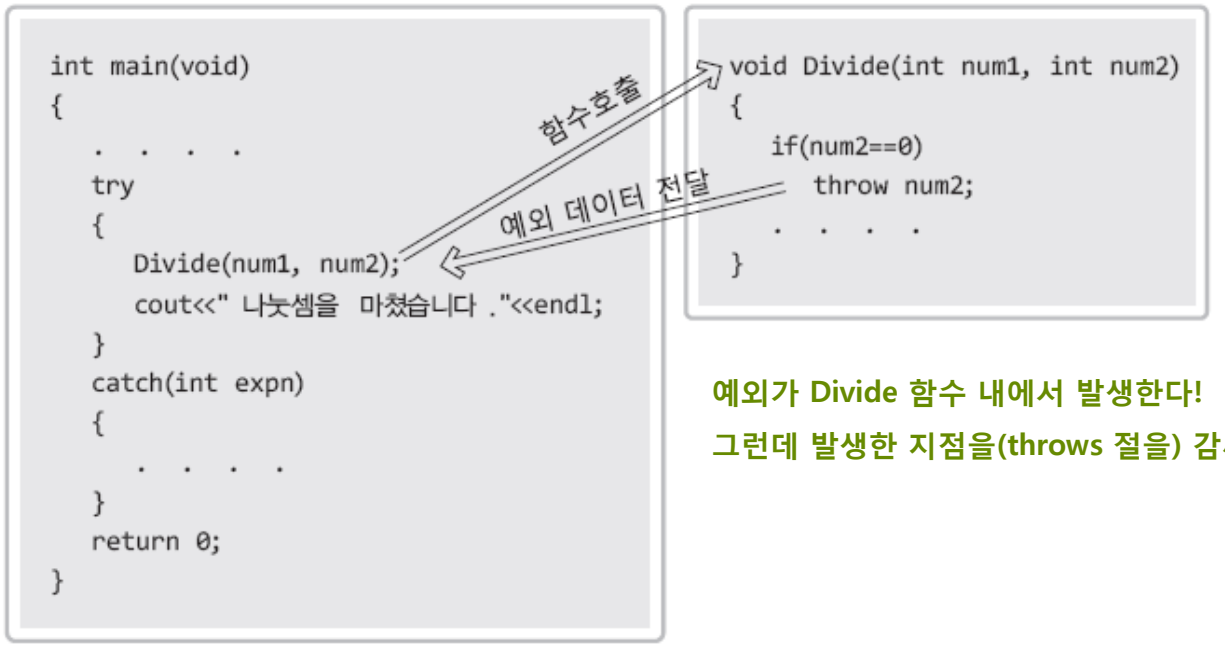
잘못 묶인 예!

```
try
{
    if(num2==0)
        throw num2;
}
catch(int expn) { . . . . }

cout<<"나눗셈의 몫: "<< num1/num2 <<endl;
cout<<"나눗셈의 나머지: "<< num1%num2 <<endl;
```

예외와 연관이 있는 부분을 모두 하나의 try 블록으로 묶어야 한다!

Stack Unwinding(스택 풀기)

예외의 전달



예외가 Divide 함수 내에서 발생한다!

그런데 발생한 지점을(throws 절을) 감싸는 try 블록이 존재하지 않는다!

예외가 처리되지 않으면, 예외가 발생한 함수를 호출한 영역으로 예외 데이터가(더불어 예외처리에 대한 책임까지) 전달된다

예외의 발생위치와 처리위치가 달라야 하는 경우

함수의 비정상 종료에 따른 처리를 main 함수에서 해야 하므로 예외의 처리는 main 함수에서 진행되어야 한다.

```
int main(void)
{
    char str1[100];
    char str2[200];

    while(1)
    {
        cout<<"두 개의 숫자 입력: ";
        cin>>str1>>str2;

        try
        {
            cout<<str1<<" + "<<str2<<" = "<<StoI(str1)+StoI(str2)<<endl;
            break;
        }
        catch(char ch)
        {
            cout<<"문자 "<< ch <<"가 입력되었습니다."<<endl;
            cout<<"재입력 진행합니다."<<endl<<endl;
        }
    }
    cout<<"프로그램을 종료합니다."<<endl;
    return 0;
}
```

```
int StoI(char * str)
{
    int len=strlen(str);
    int num=0;

    for(int i=0; i<len; i++)
    {
        if(str[i]<'0' || str[i]>'9')
            throw str[i];
        num += (int)(pow((double)10, (len-1)-i) * (str[i]+(7-'7')));
    }
    return num;
}
```

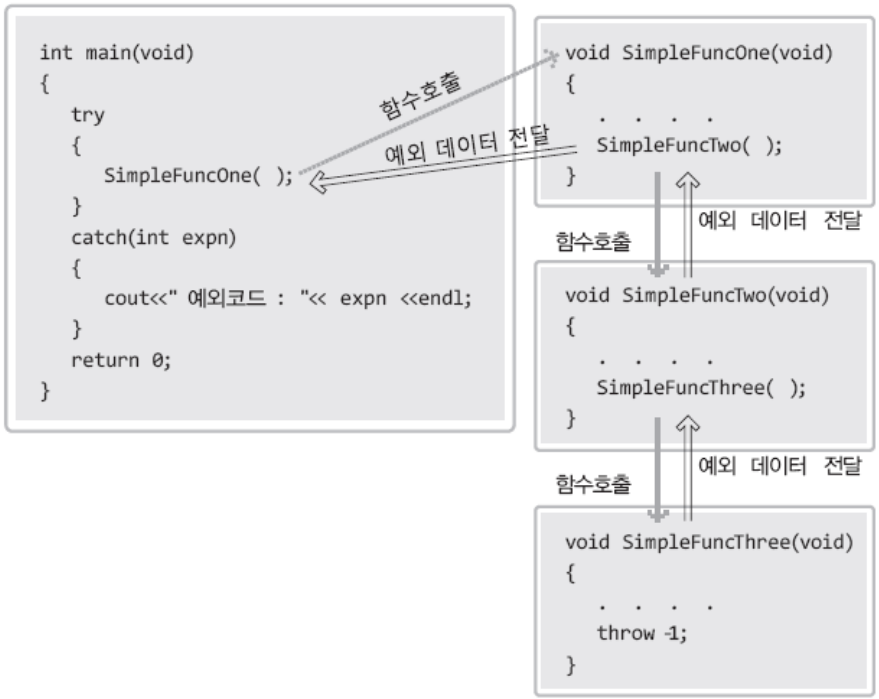
예외의 데이터가 전달되면,
함수는 더 이상 실행되지 않고 종료된다.

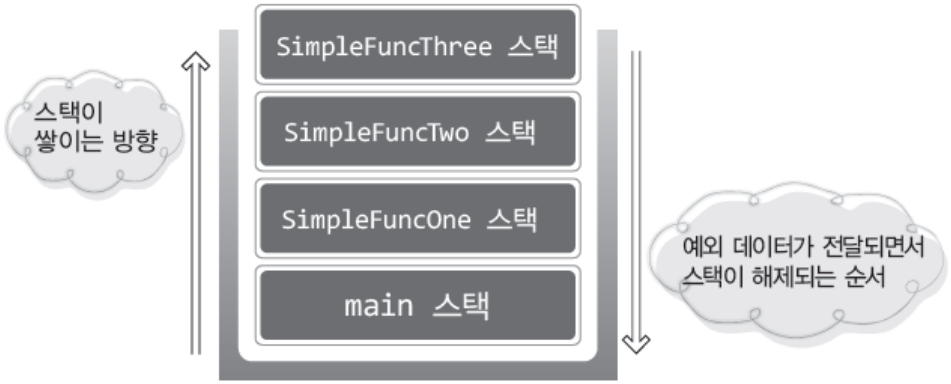
```
두 개의 숫자 입력: 123 3A5
문자 A가 입력되었습니다.
재입력 진행합니다.

두 개의 숫자 입력: 28F 211
문자 F가 입력되었습니다.
재입력 진행합니다.

두 개의 숫자 입력: 231 891
231 + 891 = 1122
프로그램을 종료합니다.
```

실행결과

스택 풀기(Stack Unwinding)

예외가 처리되지 않아서 함수를 호출한 영역으로 예외 데이터가 전달되는 현상을 가리켜 '스택 풀기'라고 한다.





main 함수에서조차 예외를 처리하지 않으면, terminate 함수(프로그램을 종료시키는 함수)가 호출되면서 프로그램이 종료됨.

StackUnwinding.cpp

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

void SimpleFuncOne(void);
void SimpleFuncTwo(void);
void SimpleFuncThree(void);

int main(void)
{
	try
	{
		SimpleFuncOne();
	}
	catch(int expn)
	{
		cout<<"예외코드: "<< expn <<endl;
	}
	return 0;
}

void SimpleFuncOne(void)
{
	cout<<"SimpleFuncOne(void)"<<endl;
	SimpleFuncTwo();
}
void SimpleFuncTwo(void)
{
	cout<<"SimpleFuncTwo(void)"<<endl;
	SimpleFuncThree();
}
void SimpleFuncThree(void)
{
	cout<<"SimpleFuncThree(void)"<<endl;
	throw -1;
}
```

자료형이 일치하지 않아도 예외 데이터는 전달

```
int SimpleFunc(void)
{
    . . . .
    try
    {
        if( . . . )
            throw -1;      // int형 예외 데이터의 발생!
    }
    catch(char expn) { . . . . }     // char형 예외 데이터를 전달하라!
    . . . .
}
```

형 변환 발생하지 않아서 예외 데이터는 SimpleFunc 함수를 호출한 영역으로 전달된다.

하나의 try 블록과 다수의 catch 블록

```
try
{
    cout<<str1<<" + "<<str2<<" = "<<StoI(str1)+StoI(str2)<<endl;
    break;
}
catch(char ch)
{
    cout<<"문자 "<< ch <<"가 입력되었습니다."<<endl;
    cout<<"재입력 진행합니다."<<endl<<endl;
}
catch(int expn)
{
    if(expn==0)
        cout<<"0으로 시작하는 숫자는 입력불가."<<endl;
    else
        cout<<"비정상적 입력이 이루어졌습니다."<<endl;

    cout<<"재입력 진행합니다."<<endl<<endl;
}
```

하나의 try 영역 내에서 종류가 다른 둘 이상의 예외가 발생할 수 있기 때문에, 하나의 try 블록에 다수의 catch 블록을 추가할 수 있다.

StackUnwinding.cpp

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

int StoI(char * str)
{
	int len=strlen(str);
	int num=0;

	if(len!=0 && str[0]=='0')
		throw 0;

	for(int i=0; i<len; i++)
	{
		if(str[i]<'0' || str[i]>'9')
			throw str[i];

		num += (int)(pow((double)10, (len-1)-i) * (str[i]+(7-
	'7')));
	}
	return num;
}

int main(void)
{
	char str1[100];
	char str2[200];

	while(1)
	{
		cout<<"두 개의 숫자 입력: ";
		cin>>str1>>str2;
		try
		{
			cout<<str1<<" + "<<str2<<" =
"<<StoI(str1)+StoI(str2)<<endl;
			break;
		}
		catch(char ch)
		{
			cout<<"문자 "<< ch <<"가 입력되었
습니다."<<endl;
			cout<<"재입력 진행합니
다."<<endl<<endl;
		}
		catch(int expn)
		{
			if(expn==0)
				cout<<"0으로 시작하
는 숫자는 입력불가."<<endl;
			else
				cout<<"비정상적 입력
이 이루어졌습니다."<<endl;

			cout<<"재입력 진행합니
다."<<endl<<endl;
		}
	}
	cout<<"프로그램을 종료합니다."<<endl;
	return 0;
}
```

전달되는 예외의 명시

```
int ThrowFunc(int num) throw (int, char)
{
    . . . .
}
```

함수 내에서 예외상황의 발생으로 인해서 int형 예외 데이터와 char형 예외 데이터가 전달될 수 있음을 명시한 선언

따라서 이 함수를 호출하는 영역의 코드는 다음과 같이 구성해야 한다.

```
try
{
    . . . .
    ThrowFunc(20);
    . . . .
}
catch(int expn) { . . . . }
catch(char expn) { . . . . }
```

int, char 이외의 예외 데이터가 전달되면, terminate 함수의 호출로 인해서 프로그램이 종료된다.
프로그램의 종료는 대비하지 못한 예외상황의 처리를 알리는 의미로 받아들여진다.

```
int SimpleFunc(void) throw ( )
{
    . . . .
}
```

어떠한 예외가 발생해도 프로그램은 종료가 된다!
이 함수는 어떠한 예외상황도 발생하지 않는다는 의미의 함수 선언!

예외상황을 표현하는 예외 클래스의 설계

예외 클래스와 예외객체

√ 예외객체 : 예외발생을 알리는데 사용되는 객체

√ 예외 클래스 : 예외객체의 생성을 위해 정의된 클래스

√ 객체를 이용해서 예외상황을 알리면 예외가 발생한 원인에 대한 정보를 보다 자세히 담을 수 있다.

입금 관련 예외상황의 표현을 위해서 정의된 클래스

```
class DepositException
{
private:
    int reqDep;   // 요청 입금액
public:
    DepositException(int money) : reqDep(money) { }
    void ShowExceptionReason() {
        cout<<"[예외 메시지: "<<reqDep<<"는 입금불가]"<<endl;   }
};
```

예외 클래스

출금 관련 예외상황의 표현을 위해서 정의된 클래스

```
class WithdrawException
{
private:
    int balance;   // 잔고
public:
    WithdrawException(int money) : balance(money) { }
    void ShowExceptionReason() {
        cout<<"[예외 메시지: 잔액 "<<balance<<", 잔액부족]"<<endl;   }
};
```

예외 클래스

예외 클래스 기반의 예외처리

```
class Account
{
private:
    char accNum[50];    // 계좌번호
    int balance;    // 잔고
public:
    Account(char * acc, int money) : balance(money) {
        strcpy(accNum, acc);  }
    void Deposit(int money) throw (DepositException) {
        if(money<0) {
            DepositException expn(money);
            throw expn;
        }
        balance+=money;
    }
    void Withdraw(int money) throw (WithdrawException)
    {
        if(money>balance)
            throw WithdrawException(balance);
        balance-=money;
    }
    void ShowMyMoney()
    {
        cout<<"잔고: "<<balance<<endl<<endl;
    }
};
```

객체 형태의 예외 데이터

임시객체의 형태로 전달 가능!

```
int main(void)
{
    Account myAcc("56789-827120", 5000);

    try
    {
        myAcc.Deposit(2000);
        myAcc.Deposit(-300);
    }
    catch(DepositException &expn)
    {
        expn.ShowExceptionReason();
    }
    myAcc.ShowMyMoney();
    try
    {
        myAcc.Withdraw(3500);
        myAcc.Withdraw(4500);
    }
    catch(WithdrawException &expn)
    {
        expn.ShowExceptionReason();
    }
    myAcc.ShowMyMoney();
    return 0;
}
```

예외객체의 멤버함수 호출

예외객체의 멤버함수 호출

ATMSim.cpp

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

class DepositException
{
private:
	int reqDep;					// 요청 입금액
public:
	DepositException(int money) : reqDep(money)
	{}
	void ShowExceptionReason()
	{
		cout<<"[예외 메시지: "<<reqDep<<"는 입금불
	가]"<<endl;
	}
};

class WithdrawException
{
private:
	int	balance;	// 잔고
public:
	WithdrawException(int money) : balance(money)
	{}
	void ShowExceptionReason()
	{
		cout<<"[예외 메시지: 잔액 "<<balance<<", 잔액부
	족]"<<endl;
	}
};
```

```
class Account
{
private:
	char accNum[50];	// 계좌번호
	int	balance;			// 잔고
public:
	Account(char * acc, int money) : balance(money)
	{
		strcpy(accNum, acc);
	}
	void Deposit(int money) throw (DepositException)
	{
		if(money<0)
		{
			DepositException expn(money);
			throw expn;
		}
		balance+=money;
	}
	void Withdraw(int money) throw (WithdrawException)
	{
		if(money>balance)
			throw WithdrawException(balance);
		balance-=money;
	}
	void ShowMyMoney()
	{
		cout<<"잔고: "<<balance<<endl<<endl;
	}
};
```

ATMSim.cpp

```
int main(void)
{
    Account myAcc("56789-827120", 5000);

    try
    {
            myAcc.Deposit(2000);
            myAcc.Deposit(-300);
    }
    catch(DepositException &expn)
    {
            expn.ShowExceptionReason();
    }
    myAcc.ShowMyMoney();

    try
    {
            myAcc.Withdraw(3500);
            myAcc.Withdraw(4500);
    }
    catch(WithdrawException &expn)
    {
            expn.ShowExceptionReason();
    }
    myAcc.ShowMyMoney();
    return 0;
}
```

상속관계에 있는 예외 클래스

```
class AccountException
{
public:
    virtual void ShowExceptionReason()=0;
};
```

```
class DepositException : public AccountException
{
private:
    int reqDep;
public:
    DepositException(int money) : reqDep(money)
    { }
    void ShowExceptionReason()
    {
        cout<<"[예외 메시지: "<<reqDep<<"는 입금불가]"<<endl;
    }
};
```

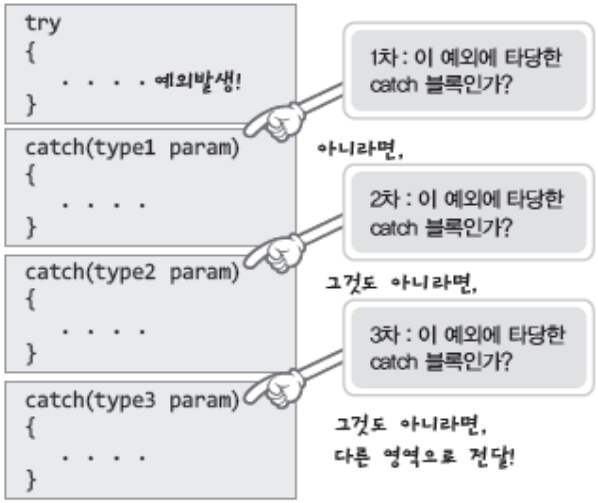
```
class WithdrawException : public AccountException
{
private:
    int balance;
public:
    WithdrawException(int money) : balance(money)
    { }
    void ShowExceptionReason()
    {
        cout<<"[예외 메시지: 잔액 "<<balance<<", 잔액부족]"<<endl;
    }
};
```

```
try
{
    myAcc.Deposit(2000);
    myAcc.Deposit(-300);
}
catch(AccountException &expn)
{
    expn.ShowExceptionReason();
}
```

```
try
{
    myAcc.Withdraw(3500);
    myAcc.Withdraw(4500);
}
catch(AccountException &expn)
{
    expn.ShowExceptionReason();
}
```

Depos~ 예외 클래스와 With~ 예외 클래스는 AccountException 클래스를 상속하므로 AccountException 을 대상으로 정의된 catch 블록에 의해 처리될 수 있다.

예외의 전달방식에 따른 주의사항



맨 위에 있는 catch 블록부터 시작해서 아래로 적절한 catch 블록을 찾아 내려온다

```
class AAA
{
public:
    void ShowYou() { cout<<"AAA exception!"<<endl; }
};

class BBB : public AAA
{
public:
    void ShowYou() { cout<<"BBB exception!"<<endl; }
};

class CCC : public BBB
{
public:
    void ShowYou() { cout<<"CCC exception!"<<endl; }
};
```

예외객체의 전달과정에서의 문제점은?

```
try
{
    ExceptionGenerator(3);
    ExceptionGenerator(2);
    ExceptionGenerator(1);
}
catch(AAA& expn)
{
    cout<<"catch(AAA& expn)"<<endl;
    expn.ShowYou();
}
catch(BBB& expn)
{
    cout<<"catch(BBB& expn)"<<endl;
    expn.ShowYou();
}
catch(CCC& expn)
{
    cout<<"catch(CCC& expn)"<<endl;
    expn.ShowYou();
}
```

CatchFlow.cpp

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

class AAA
{
public:
	void ShowYou() { cout<<"AAA exception!"<<endl; }
};

class BBB : public AAA
{
public:
	void ShowYou() { cout<<"BBB exception!"<<endl; }
};

class CCC : public BBB
{
public:
	void ShowYou() { cout<<"CCC exception!"<<endl; }
};

void ExceptionGenerator(int expn)
{
	if(expn==1)
		throw AAA();
	else if(expn==2)
		throw BBB();
	else
		throw CCC();
}
```

```
int main(void)
{
	try
	{
		ExceptionGenerator(3);
		ExceptionGenerator(2);
		ExceptionGenerator(1);
	}
	catch(AAA& expn)
	{
		cout<<"catch(AAA& expn)"<<endl;
		expn.ShowYou();
	}
	catch(BBB& expn)
	{
		cout<<"catch(BBB& expn)"<<endl;
		expn.ShowYou();
	}
	catch(CCC& expn)
	{
		cout<<"catch(CCC& expn)"<<endl;
		expn.ShowYou();
	}
	return 0;
}
```

예외처리와 관련된 또 다른 특성들

new 연산자에 의해서 전달되는 예외

```
int main(void)
{
    int num=0;
    try
    {
        while(1)
        {
            num++;
            cout<<num<<"번째 할당 시도"<<endl;
            new int[10000][10000];
        }
    }
    catch(bad_alloc &bad)
    {
        cout<<bad.what()<<endl;
        cout<<"더 이상 할당 불가!"<<endl;
    }
    return 0;
}
```

오늘날 C++표준에서는 new연산자가 메모리할당에 실패

==> 헤더파일 new에 선언되어있는 bad_alloc 예외처리가 전달된다고 명시됨

실행결과

```
1번째 할당 시도
2번째 할당 시도
3번째 할당 시도
4번째 할당 시도
5번째 할당 시도
bad allocation
더 이상 할당 불가!
```

bad_alloc과 같이 프로그래머가 정의하지 않아도 발생하는 예외도 있다.

그리고 Chapter 16에서는 이러한 유형의 예외 중 하나로, 형 변환 시 발생하는 bad_cast 예외를 소개한다.

모든 예외를 처리하는 catch 블록

```
try
{
    . . . .
}
catch(...)     // ... 은 전달되는 모든 예외를 다 받아주겠다는 선언
{
    . . . .
}
```

마지막 catch 블록에 덧붙여지는 경우가 많은데, 대신 catch의 매개변수 선언에서 보이듯이,
발생한 예외와 관련해서 그 어떠한 정보도 전달받을 수 없으며, 전달된 예외의 종류도 구분이 불가능하다.

예외 던지기

catch 블록에 전달된 예외는 다시 던져질 수 있다.

그리고 이로 인해서 하나의 예외가 둘 이상의 catch 블록에 의해서 처리되게 할 수 있다.

```
void Divide(int num1, int num2)
{
    try
    {
        if(num2==0)
            throw 0;
        cout<<"몫: "<<num1/num2<<endl;
        cout<<"나머지: "<<num1%num2<<endl;
    }
    catch(int expn)
    {
        cout<<"first catch"<<endl;
        throw;      // 예외를 다시 던진다!
    }
}
```

```
int main(void)
{
    try
    {
        Divide(9, 2);
        Divide(4, 0);
    }
    catch(int expn)
    {
        cout<<"second catch"<<endl;
    }
    return 0;
}
```

실행결과

```
몫: 4
나머지: 1
first catch
second catch
```

Q & A