24장. 예외 처리

01_ 예외 처리의 기본

02_ 구조적 예외 처리 제대로 사용하기

예외 처리(Exception Handling)

- 예외란 우리가 당연하다고 가정한 상황이 거짓이 되는 경우를 말한다.
  - 대표적인 예외의 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 것을 말한다.
  - 예) 컴퓨터에 사용 가능한 메모리가 부족해서 동적 메모리 할당이 실패한 경우에 예외 처리를 잘 하지 못한 프로그램은 비정상 종료할 수 있다. 반면에 예외 처리를 잘 한 프로그램은 사용자가 작업 중이던 정보를 잘 보관한 후에 정상적으로 종료할 수 있다.

DynamicArray 클래스 다시 보기(1)

- 예외 처리를 연습하기 위해서 수정한 동적 배열 클래스

```
class DynamicArray
{
public:
          // 생성자, 소멸자
          DynamicArray(int arraySize);
          ~DynamicArray();

          // 접근자
          void SetAt(int index, int value);
          int GetAt(int index) const;
          int GetSize() const;


protected:
          int* arr; // 할당한 메모리 보관
          int size; // 배열의 길이 보관
};
```

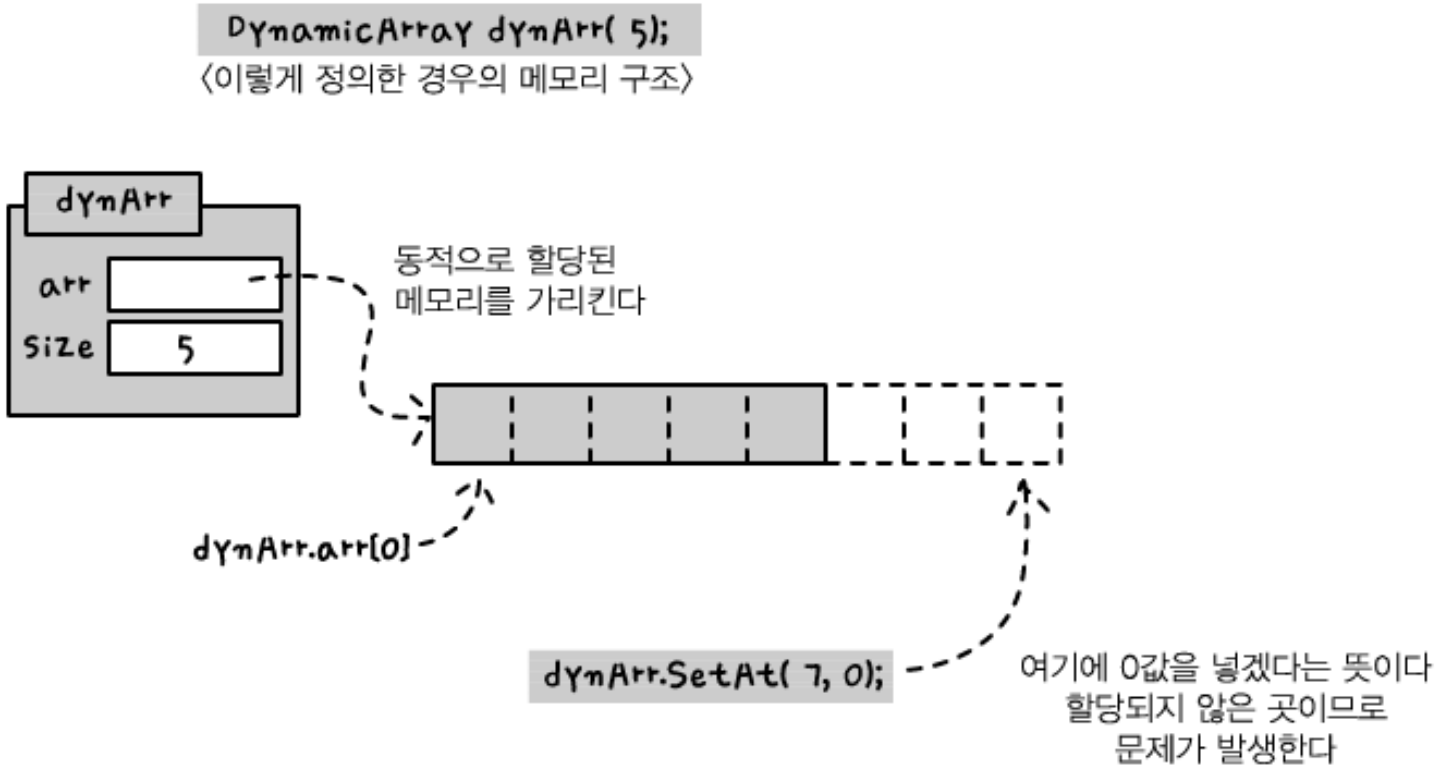
DynamicArray 클래스 다시 보기(2)

- 예외 처리를 연습하기 위해서 수정한 동적 배열 클래스

```
DynamicArray::DynamicArray(int arraySize)
{
        arr = new int [arraySize];
        size = arraySize;
}

DynamicArray::~DynamicArray()
{
        delete[] arr;
}

// 원소의 값을 바꾼다
void DynamicArray::SetAt(int index, int value)
{
        arr[index] = value;
}

// 원소의 값을 반환한다
int DynamicArray::GetAt(int index) const
{
        return arr[index];
}

// 배열의 길이를 반환한다.
int DynamicArray::GetSize() const
{
        return size;
}
```

DynamicArray 클래스의 문제점

- SetAt() 함수나 GetAt() 함수에 너무 큰 인덱스를 넘겨주면 할당되지 않은 메모리를 건드려서 프로그램이 비정상 종료 할 수 있다.



반환 값을 사용한 예외 처리(1)

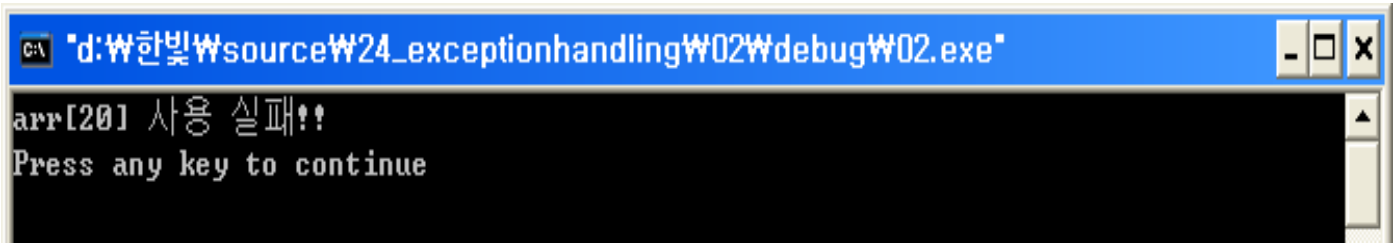
- SetAt() 함수가 실패했을 때 반환 값으로 알려주는 예

```
// 원소의 값을 바꾼다
// 인덱스의 범위가 잘못된 경우 false 반환
bool DynamicArray::SetAt(int index, int value)
{
        // 인덱스를 확인한다.
        if (index < 0 || index >= GetSize() )
                return false;

        arr[index] = value;
        return true;
}
```

```
// 크기가 10인 배열 객체를 만든다.
DynamicArray arr(10);

// 올바른 인덱스를 참조한다
bool b;
b = arr.SetAt(5, 0);
if (!b)
        cout << "arr[5] 사용 실패!!\n";

// 일부러 범위 밖의 인덱스를 참조한다
b = arr.SetAt(20, 0);
if (!b)
        cout << "arr[20] 사용 실패!!\n";
```

반환 값을 사용한 예외 처리(2)

- 실행 결과



- 반환 값을 사용한 예외 처리의 문제점
  - 본연의 소스 코드와 예외 처리 코드가 뒤엉켜서 지저분하고 읽기 어렵다.
  - 예외 처리 때문에 반환 값을 본래의 용도로 사용할 수 없다.

구조적 예외 처리(1)

- C++의 구조적 예외 처리를 사용해서 실패를 알리는 예

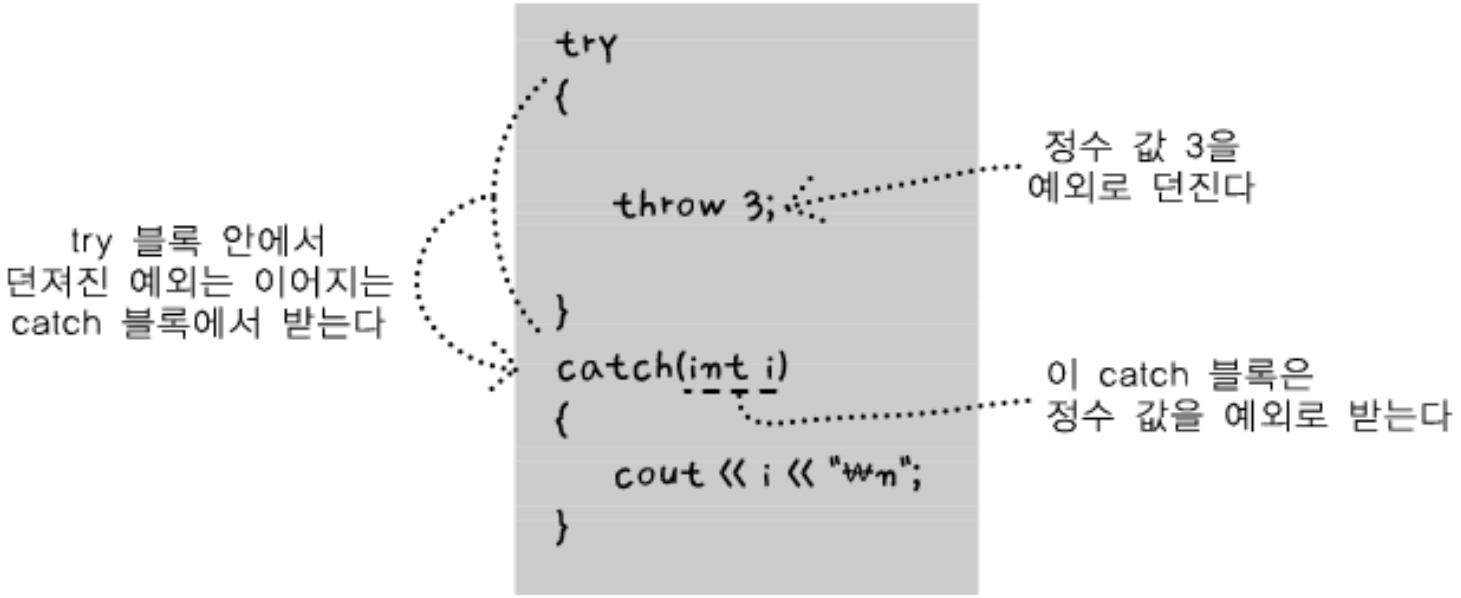
```
// 원소의 값을 바꾼다
void DynamicArray::SetAt(int index, int value)
{
          // 인덱스의 범위가 맞지 않으면 예외를 던진다
          if (index < 0 || index >= GetSize())
                    throw "Out of Range!!!";

          arr[index] = value;
}
```

```
// 크기가 10인 배열 객체를 만든다.
DynamicArray arr(10);

try
{
          arr.SetAt(5, 100);
          arr.SetAt(8, 100);
          arr.SetAt(10, 100);
}
catch(const char* ex)
{
          cout << "예외 종류 : " << ex << "\n";
}
```

구조적 예외 처리(2)

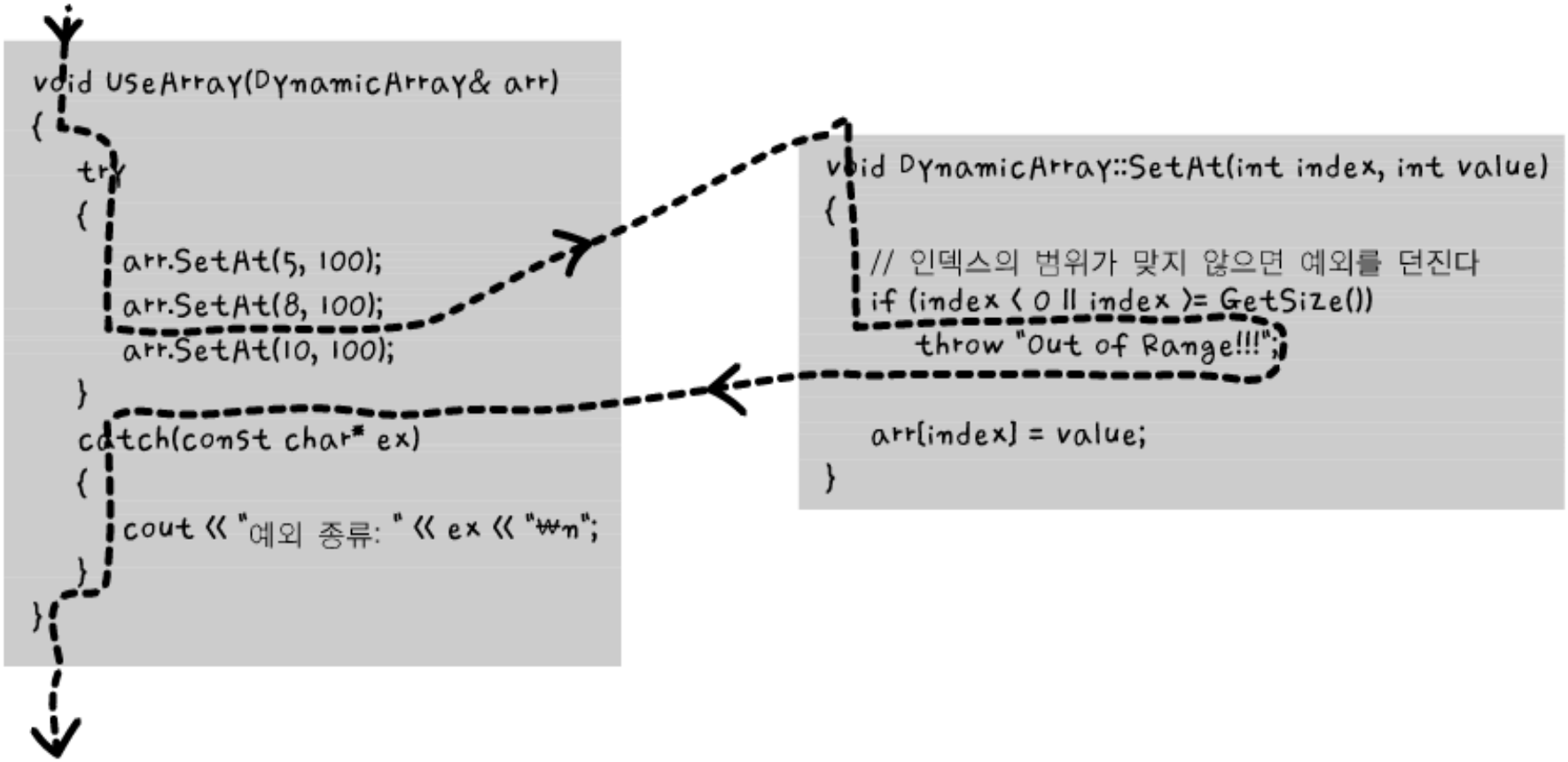
- 실행 결과

- throw, try, catch의 사용



구조적 예외 처리(3)

- 예외의 발생과 실행의 흐름



예외 객체의 사용(1)

- 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;
}
```

예외 객체의 사용(2)

- 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(1)

- 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;
}
```

구조적 예외 처리의 규칙(2)

- 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.

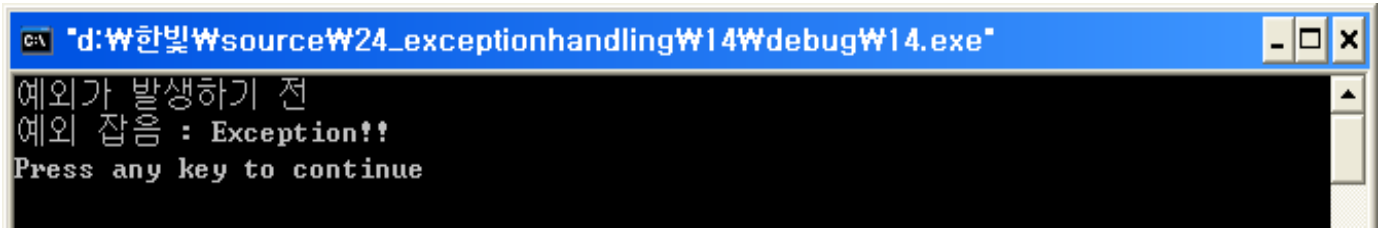
리소스의 정리(1)

- 리소스를 정리하기 전에 예외가 발생한 경우의 문제점 확인

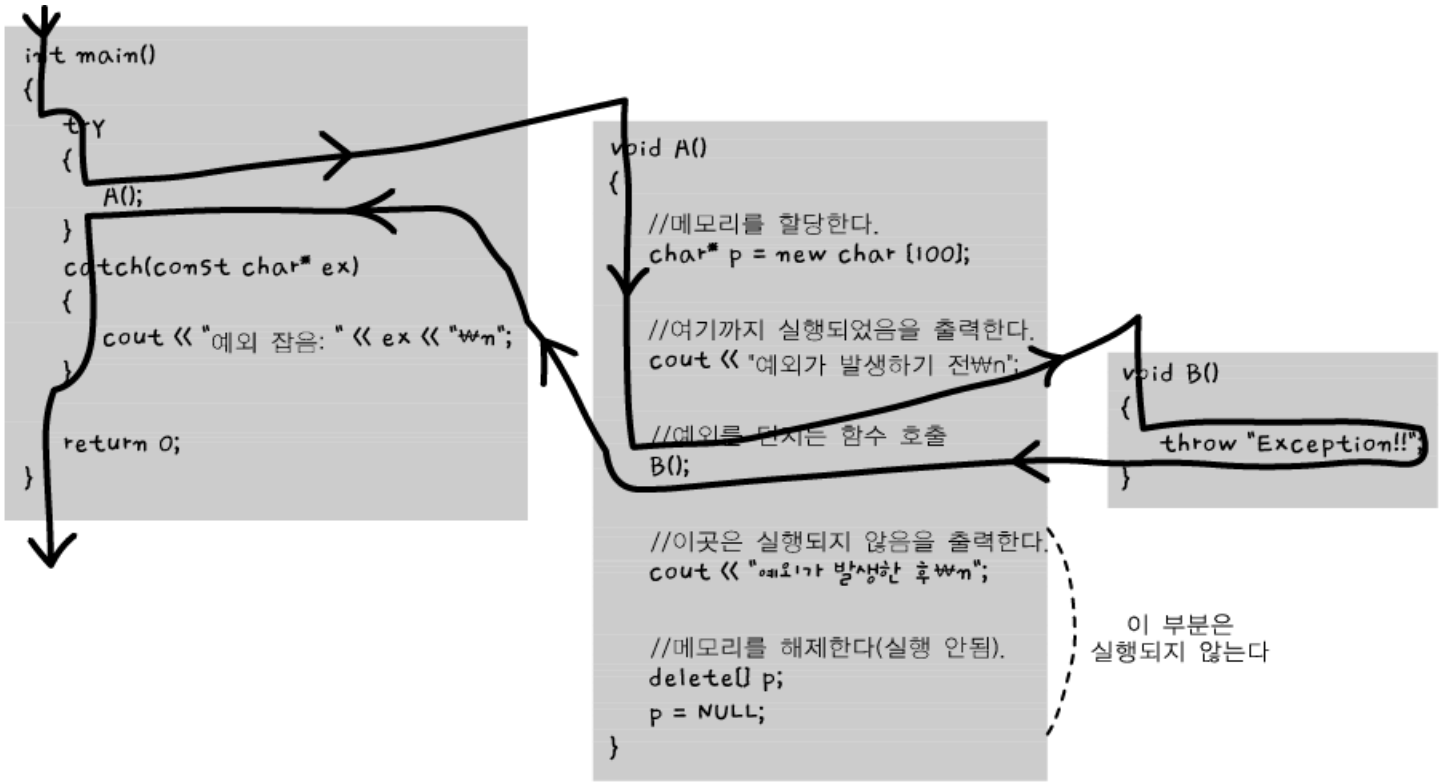
```
try
{
		// 메모리를 할당한다.
		char* p = new char [100];

		// 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";
}
```

리소스의 정리(2)

- 실행 결과



- 실행의 흐름



소멸자를 사용한 리소스 정리(1)

- 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 사용한 리소스 정리(2)

- 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";
}
```

소멸자를 사용한 리소스 정리(3)

- 실행 결과

생성자에서의 예외 처리

- 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.
        }
}
```

소멸자에서의 예외 처리

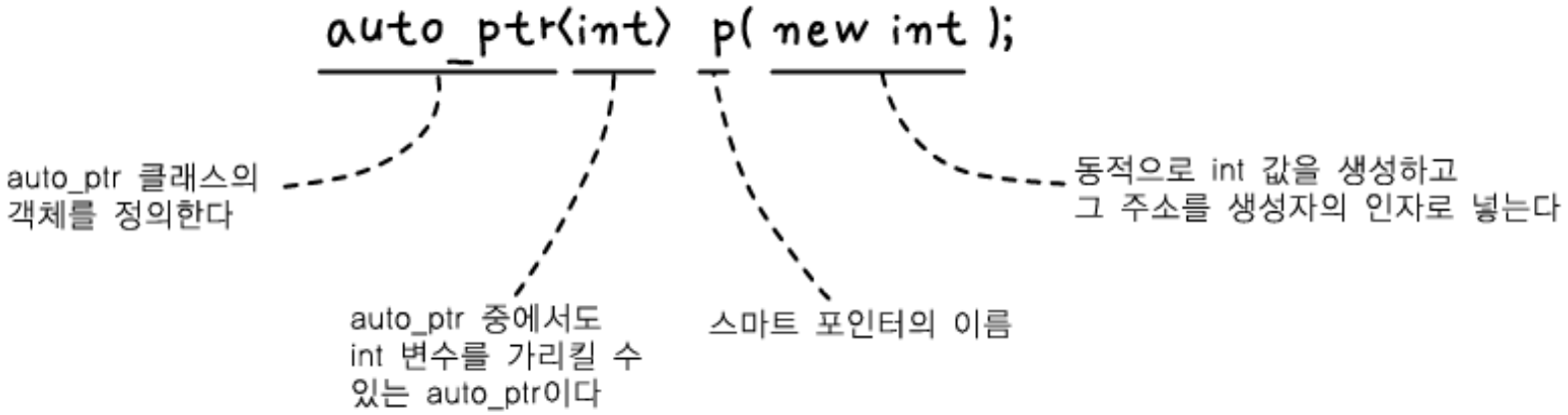
- 소멸자에서 예외가 발생하는 경우에는 프로그램이 비정상 종료할 수 있으므로, 소멸자 밖으로 예외가 던져지지 않게 막아야 한다.

```
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

Q&A