고급 객체 지향 프로그래밍 제15장

강대기

May 14, 2008

프렌드

내포 클래스

예외

RTTI - 실행 시간 데이터형 정보

데이터형 변환 연산자

프렌드에는 무엇이 있나?

- 프렌드 함수 - 클래스를 위한 확장 인터페이스의 일부.

프렌드에는 무엇이 있나?

- 프렌드 함수 - 클래스를 위한 확장 인터페이스의 일부.
- 프렌드 클래스 - 오리지널 클래스의 private/protected에 접근.

프렌드에는 무엇이 있나?

- 프렌드 함수 - 클래스를 위한 확장 인터페이스의 일부.
- 프렌드 클래스 - 오리지널 클래스의 private/protected에 접근.
- 프렌드 멤버 함수.

프렌드는 객체 지향 철학을 위반하는가?

- 프렌드 관계는 바깥에서 설정할 수 없음.
- 프렌드들에게 클래스의 private 부분에 대한 접근을 허용하더라도 객체 지향 철학을 위반하는 건 아님.
- 오히려 public 인터페이스에 융통성을 제공함.

프렌드 클래스란?

- 예: 텔레비전과 리모트의 간단한 시뮬레이션 (1000쪽).
  - 서로 ISA 관계나 has-a 관계는 성립하지 않음.
  - 그런데, 리모트는 텔레비전의 상태를 변경할 수 있음.
  - *상속이나 내포에 적합하지 않으면서도 종속 관계이므로, 프렌드로 만들면 적당함.*
    - Remote 클래스가 Tv 클래스의 프렌드
- 1001 쪽 예제 - Tv.h
- Tv 클래스 내부에서 Remote 클래스를 프렌드 선언 (1001쪽), 또는 사전 선언(forward declaration) (1006쪽)
- 예: 텔레비전과 리모트의 간단한 시뮬레이션 (1000쪽).
  - private/protected/public 어느 위치건 상관 없음
  - friend class Remote;

프렌드 클래스가 아니었다면?

- 프렌드 클래스가 이러한 관계를 표현하는 데 자연스러운 방법.
- 프렌드 클래스가 없었다면
  - Tv 클래스의 private 부분들을 public으로 만들거나
  - 텔레비전과 리모트가 함께 들어있는 다루기 어렵고 덩치가 큰 클래스를 만들어야 함
  - 또한 하나의 리모트로 여러 대의 Tv를 제어할 수 있다는 사실을 반영하지 못함

프렌드 멤버 함수란?

- 특정 멤버 함수 하나만 프렌드로 하려면?
- 그러고 보니 (너무 우연히도) 1002쪽에서 set_chan 메쏘드만 private 멤버에 접근하고 있음
- 프렌드 멤버 함수가 출동하면 어떨까?

```
class Tv
{
  friend void Remote::set_chan(Tv& t, int c);
  ...
}
```

프렌드 멤버 함수를 위한 사전 선언

- Tv에서 Remote 멤버 함수를 지정하려면 컴파일러가 이미 Remote 클래스를 알아야 함. 그런데, Remote의 멤버 함수에서는 Tv 클래스의 멤버들을 이미 사용하고 있음 → 프렌드가 아닌 클래스를 사전 선언함으로써 해결!
- 다음과 같이 사전 선언을 사용함

```
class Tv;
class Remote {...};
class Tv {...};
```

프렌드 멤버 함수를 위한 사전 선언

- 다음과 Tv와 Remote의 위치를 바꾸면 컴파일러가 Tv를 처리할 때, Remote의 멤버 함수를 모르므로 컴파일 에러 발생함.

```
class Remote;
class Tv {...}; // 에러 발생
class Remote {...};
```

프렌드 멤버 함수를 위한 사전 선언

- 그런데, 처음의 경우에도 Remote 클래스를 처리할 때, Tv의 멤버 함수들은 컴파일러가 모르는 상황 → Remote를 메쏘드 선언으로 제한함 (1008–1009쪽)

```
class Tv; // 사전 선언
class Remote {...}; // Remote 메쏘드들의 원형만 선언
class Tv {...};
// Remote 메쏘드들의 정의들
inline bool Remote::volup(Tv& t) {return t.volup();}
...
```

- 인라인 함수는 내부 링크이므로, 헤더에 넣음

클래스 프렌드와 클래스 멤버 프렌드의 관계

- 1010쪽의 그림에 잘 나와 있음
- Remote 클래스 전체를 프렌드로 만든다면 사전 선언은 필요 없음

```
friend class Remote;
```

상호 프렌드 (mutual friend)

- Tv와 양방향 통신할 수 있는 대화형 리모트가 있다면? → 서로 프렌드로 함(1011쪽)

```
class Tv {
  friend class Remote;
public:
  void buzz(Remote& r);
...
}
class Remote {
  friend class Tv;
public:
  void bool volup(Tv& t) {t.volup();};
...
}
inline void Tv::buzz(Remote& r) { ... }
```

상호 프렌드 (mutual friend)

- Remote 선언이 Tv 선언 뒤에 나오므로 Remote::volup()을 Remote 클래스 안에서 정의했음.
- Tv::buzz() 메쏘드 정의는 Remote 선언 뒤에 와야 하므로, Tv 선언 바깥에 정의함.
- Tv::buzz() 메쏘드 정의를 인라인으로 안해도 된다면, 별개의 메쏘드 정의 파일 (.cpp)에 정의해도 됨.

공유 프렌드

- 하나의 함수가 서로 다른 두 클래스에 들어있는 private 데이터에 접근하고 싶다면?
  - 원칙적으로 하나의 클래스의 멤버 함수를 프렌드로 함
  - 때로는 하나의 외부 함수가 두 개의 클래스의 프렌드로 하는 게 더 이치에 맞을 경우가 있음
  - 예: Probe와 Analyzer의 시간을 동기화하는 경우

```
class Analyzer; //  사전 선언
class Probe {
  friend void sync(Analyzer& a, const Probe& p); // a를 p에 맞춤
  friend void sync(Probe& p, const Analyzer& a); // p를 a에 맞춤
... }
class Analyzer {
  friend void sync(Analyzer& a, const Probe& p); // a를 p에 맞춤
  friend void sync(Probe& p, const Analyzer& a); // p를 a에 맞춤
... }
inline void sync(Analyzer& a, const Probe& p) { ... } // 프렌드 함수 정의
inline void sync(Probe& p, const Analyzer& a) { ... } // 프렌드 함수 정의
```

내포 클래스(nested class)란?

- 클래스 선언을 다른 클래스 안에 내포시킴
- 내포된 새로운 데이터 형에 대해 클래스 사용 범위를 제공 → 이름이 난잡해지지 않음(1013쪽 아래부분 코드, 781쪽, 그리고 1014쪽 아래부분 코드)
- 내포 클래스를 가진 클래스의 멤버 함수는 내포 클래스 객체를 생성하여 사용할 수 있음
- 바깥에선 내포 클래스 선언이 public 부분에 들어있고, 사용 범위 결정 연산자(::)를 사용해야만 사용할 수 있음

내포 클래스(nested class)와 컨테인먼트

- 컨테인먼트는 어떤 클래스 객체를 다른 클래스의 멤버로 가지는 것
- 내포 클래스는 클래스 멤버를 생성하지 않고, 대신 내포 클래스 선언을 내포하는 클래스에서만 지역적으로 알려지는 하나의 데이터 형을 정의함

내포 클래스(nested class)를 사용하는 이유

- 다른 클래스의 구현을 지원
- 이름 충돌을 막음

내포 클래스(nested class)를 사용 방법

- Queue 클래스 안의 Node 구조체 (1013쪽 아래 부분 코드)
- Queue 클래스 안의 enqueue 함수에서만 Node 구조체가 사용됨을 확인 (1014쪽 처음 코드)
- → Queue 클래스 안의 Node 클래스 구현 (1014쪽 마지막 코드)
- → Queue 클래스 안의 enqueue 함수 변경 (1015쪽 코드)
- 생성자를 클래스 선언 파일(.h)이 아닌 메쏘드 정의 파일(.cpp)에 넣고자 한다면?

```
Queue::Node::Node(const Item &i) : item(i), next(0) {}
```

내포 클래스(nested class)에 대한 두 종류의 접근

1. 내포 클래스가 선언된 장소가 내포 클래스의 사용 범위를 제한함 (사용 범위)
   - 내포 클래스에 대한 사용 범위는 내포 구조체 (nested struct), 내포 열거체 (nested enum)에 대해서도 동일하게 적용
2. 내포 클래스의 public, protected, private 부분도 그 클래스 멤버에 대한 접근을 제한함 (접근 제어)

사용 범위

- 내포 클래스 Node가 내포한 클래스(제 2 클래스) Queue의 private 부분에 선언된 경우
  - Queue 클래스 멤버들은 Node 객체 사용 가능하나, 바깥 세계의 다른 부분에선 Node 클래스의 존재도 모름
  - Queue 클래스의 파생 클래스도 Node 클래스를 사용할 수 없음
- 내포 클래스 Node가 내포한 클래스(제 2 클래스) Queue의 protected 부분에 선언된 경우
  - Queue 클래스 멤버들은 Node 객체 사용 가능하나, 바깥 세계의 다른 부분에선 Node 클래스의 존재도 모름
  - Queue 클래스의 파생 클래스는 Node 클래스를 사용할 수 있음

public 부분에 선언된 내포 클래스

- 내포 클래스 Node가 내포한 클래스(제 2 클래스) Queue의 public 부분에 선언된 경우
  - Queue 클래스 멤버들은 Node 객체 사용 가능하고, 바깥 세계의 다른 부분에서도 Node 클래스를 사용할 수 있음
  - Queue 클래스의 파생 클래스도 Node 클래스 사용 가능

```
class Team { public: class Coach { ... }; ... }
```

- Queue 클래스의 파생 클래스도 Node 클래스를 사용할 수 있음 (예: `Team::Coach forhire;`)
- public으로 선언된 내포 열거체 → 모두 쓸 수 있는 클래스 상수

public 열거체를 통해 외부 세계에 클래스 상수 제공

- 다음과 같은 형태

  `public: enum { ... }`

- 1234 쪽의 표 17.1 — 출력 형식 지정 상수들
- 1237 쪽의 표 17.2 — `setf(long, long)`을 위한 전달 인자들
- 1241 쪽의 표 17.3 — 표준 조정자들
- 1280 쪽의 표 17.8 — 파일 모드 상수들
- 1282 쪽의 표 17.9 — C++와 C의 파일 열기 모드

접근 제어

- 내포(된) 클래스에 대해서 내포한 클래스가 더 특별한 접근 권한을 가지지는 않으며, 내포 클래스도 내포한 클래스에 대해 더 특별한 접근 권한을 가지지 않음
- Queue 클래스 객체는 Node 객체의 public 멤버들에 대해서만 명시적으로 접근 가능
  - 그래서 일반적으로 내포 클래스 자체는 private 영역에 선언되고, 내포 클래스의 데이터 멤버들은 public 으로 선언
  - 데이터 멤버는 private 으로 하는 일반적 관행에 위배되나, 내포 클래스는 private에 있으므로 바깥 세계엔 보이지 않음
  - 즉, Queue 클래스의 메쏘드들은 Node 멤버들에 직접 접근 가능하지만, Queue 클래스를 사용하는 객체나 함수들은 Node 멤버들에 접근 불가능

내포 클래스,내포 구조체,내포 열거체의 사용 범위 특성

Table: 1017쪽의 표 15.1 — 내포 클래스, 내포 구조체, 내포 열거체의 사용 범위 특성

제 2 클래스에서 선언된 장소	제 2 클래스에서 사용 여부	제 2 클래스로부터 파생된 클래스에서 사용 여부	바깥 세계에서 사용 여부
private 부분	가능	불가능	불가능
protected 부분	가능	가능	불가능
public 부분	가능	가능	클래스 제한자를 사용하여 가능

템플릿에서의 내포

- 클래스 템플릿에서도 내포 클래스의 사용이 가능함 → 1018 쪽의 queuetp.h
- 데이터 형 매개 변수 Item 이 내포 클래스에서도 사용 가능함
  - QueueTp<double> dq;는 Node가 double 형을 받아들이게 하며, Node는 QueueTp<double>::Node 형으로 정의됨
  - QueueTp<char> dq;는 Node가 char 형을 받아들이게 하며, Node는 QueueTp<char>::Node 형으로 정의됨
- queuetp.h를 테스트하는 프로그램은 1020 쪽의 nested.cpp

예외란?

- C++의 예외 처리란 프로그램을 먹통으로 만드는 특별한 상황에 대처하는 메커니즘
  - 사용할 수 없는 파일을 열려고 하거나
  - 사용 가능한 메모리보다 더 많은 양의 메모리를 요구하거나
  - 처리할 수 없는 값들을 만날 때
  - 그 외 프로그래머가 예상치 못하는 사태들

예외 사태를 어떻게 처리할 것인가?

- abort() 호출 (1023 쪽)
- 에러 코드 리턴 (1024 쪽)
- 예외 메커니즘 (1027 쪽)
  - 예외로 기본형 데이터 사용 (1027 쪽)
  - 예외로 객체 사용 (1031 쪽)
    - 일반 객체 (1031 쪽)
    - exception 클래스 (1047 쪽)

예외는 동일한 제어 방식을 제공함

- 예를 들어 조화평균 $2.0 \times x \times y/(x+y)$
  - 어떤 컴파일러는 무한대를 나타내는 부동 소숫점 생성 - `Inf`, `inf`, `INF`
  - 어떤 컴파일러는 먹통이 되는 프로그램 생성
- 최선의 방법은 모든 시스템에서 동일한 제어 방식으로 행동하는 코드 작성

abort() 호출

- 한 전달 인자가 다른 전달 인자의 부정이면 `abort()` 호출
- `abort()` 원형은 `cstdlib`에 있음
- 표준 에러 스트림(cerr)에 “abnormal program termination” 식의 메시지를 출력하고 그 자리에서 바로 종료
- 운영 체제나 모 프로그램에 특정 값을 반환
- 파일 버퍼를 비울 수도 있고 안비울 수도 있음
- 예제 – 1023쪽

에러 코드 리턴

- 함수의 리턴 값을 통해 문제를 파악함
- 예를 들어 ostream 클래스의 get(void) 멤버는 파일 끝을 만나면 EOF를 리턴
- 예제 – 1025쪽
- hmean()을 bool 함수로 다시 정의하고, 결과는 double * 매개 변수로 받음
- 포인터가 더 가독성이 좋으므로 참조보다는 포인터를 사용함
- 리턴 값이 아닌 전역 변수를 사용할 수도 있음 – C의 math 라이브러리의 errno 전역 변수

예외 메커니즘

- 프로그램이 실행하는 도중에 발생하는 예외적인 상황에 대한 문법적인 응답
  1. 예외를 발생시키는 부분
  2. 예외 핸들러를 사용하여 예외를 포착하는 부분
  3. try-catch 블럭을 사용하는 부분
- 참고로 정통 C++는 finally 가 없으나 MS Visual C++는 가지고 있음

예외 메커니즘 (1028 쪽의 프로그램)

- throw 키워드는 예외의 발생을 나타내며, 기본형이나 객체가 뒤따를 수 있음
- catch 키워드는 예외의 포착을 나타내며, 뒤의 소괄호에 예외의 데이터 형이 나오고, 그 뒤에 코드 블럭이 나옴
- try 블럭은 예외들이 발생할 수 있는 하나의 코드 블럭으로, 발생한 예외를 처리하기 위한 하나 이상의 catch 블럭들이 뒤따름
- 예외 발생은 그 함수 실행을 종료시키지만, 호출한 함수가 아니라 해당 예외를 처리하는 try-catch 블럭을 가진 가장 최근에 호출한 함수로 제어를 넘김
- 예외 발생 시 try 블럭이 없거나 일치하는 데이터 형이 없으면 abort() 함수 호출

예외로 객체 사용하기

- 일반적으로 예외를 발생시키는 함수는 객체를 발생시킴 — 서로 다른 예외 데이터 형으로 다른 함수나 예외 상황과 구별
- 예외가 던질 bad_hmean 객체(1031 쪽 아래 부분)에 대한 사용 예

```
if (a==-b) throw bad_hmean(a,b);
```

- 예외 지정 (exception specification) 으로 함수 정의를 한정지음

```
double hmean(double a, double b) throw (bad_hmean);
double hmean(double a, double b) throw (bad_hmean)
{
}
```

예외 지정

- 예외 지정은 함수가 발생시키는 예외의 종류를 컴파일러에게 알려줌
  - 다른 데이터 형의 예외를 전달하면 abort() 함수 호출
- 예외 지정의 사용은 그 함수를 읽는 사람에게 그 함수가 예외를 전달한다는 사실을 알려줌
  - 그 사람이 try-catch 블럭을 제공할 수 있음을 알게 함
- 여러 예외들에 대한 예외 지정 (exception specification)

```
double multi_err(double z) throw (const char*, double);
```

- 예외 지정에 비어있는 괄호를 사용하면 그 함수가 예외를 발생시키지 않는다는 의미

```
double simple(double z) throw ();
```

예외 지정

- 하나의 try 블럭에서 두 개의 예외를 같이 받을 수 있음

```
try {
  ...
} catch (bad_hmean& bh) { // hmean 함수
  ...
} catch (bad_gmean& bg) { // gmean 함수
  ...
}
```

- 1034쪽을 보면, bad_hmean 과 bad_gmean 이 다른 방식으로 예외 메시지를 보냄
- 1035쪽을 보면, main() 함수가 bad_hmean 과 bad_gmean 을 다른 방식으로 처리함

스택 풀기

- 예외가 호출된 함수에서 발생해도 그 함수에 try-catch 블럭이 없다면, 그 블럭을 가지고 있으면서 호출 정보를 저장하는 스택의 관점에서 가장 가까운 호출한 함수에게 제어가 전달됨
- 스택에 쌓여있는 호출 정보를 풀어가면서 이러한 탐색 작업을 수행함 (Java에선 `e.printStackTrace` 가 이러한 스택 풀기의 내용을 보여줌)
- 1038쪽 그림 15.3에 잘 나와있음
- 던지면서 지역 변수들은 그래도 제대로 파괴해 줌 → 그러나 최선을 다해 예외 안전성을 확보해야 함(EC++ 항목 29, MEC++ 항목 9–15)

함수는 예외를 받아서 처리하고 다시 던지거나, 그냥 던질 수 있음

- 받아서 처리하고 다시 던짐

```
catch (bad_hmean& bh) {
  bh.mesg();
  std::cout << "means()에서 잡혔다!";
  throw;  // 다시 던짐
}
```

- 무시하고 그냥 던짐 → 따라서 단순히 그 함수의 예외 말고 그 함수가 호출하는 함수의 모든 예외도 지정해야 함 (그렇지 않을 경우, 1059쪽의 기대하지 않은 예외에 속함)

```
double means(double a, double b) throw (bad_hmean, bad_gmean)
{ ... }
```

예외의 그 밖의 기능

- 반복하듯이, 예외는 try-catch 블럭이 있는 최초의 함수까지 실행을 계속 옮김
- 예외는 참조로 받는 게 원칙임 (MEC++ 항목 13)
  - 1044 쪽의 코드 → oops를 지역 변수이므로 이미 존재하지 않음. 따라서 원칙적으로 복사본을 만들어서 날리는 데, 만일 그걸 값으로 받으면 결국 두 번 복사되는 것임.
  - 업캐스팅이 가능하므로 파생 클래스의 데이터나 멤버 함수에 접근 가능함
- 예외 클래스 파생 순서의 반대로 catch 블럭을 놓아야 예외가 제대로 잡힘. 그렇지 않으면 기초 클래스가 다 잡음

예외의 그 밖의 기능

- 어떤 예외라도 포착할 수 없을까?
  - 사용자가 어떤 예외 데이터 형을 기대하는지 모를 때
  - 내용을 정확히 모르는 다른 함수를 호출하는 함수를 작성할 때
  - → `catch (...) { 명령문들 }` 사용 (switch 의 default 문과 비슷)
- 반복하자면, 참조 대신 객체를 포착한다면 기초 클래스 부분만 남으므로 참조로 해야 함

exception 클래스

- 예외의 주 목적 — 에러 처리를 프로그램 설계 자체에 반영하여, 예외 시스템을 통해 에러 처리가 융통성 있고 간편해짐
- exception 헤더 파일에 exception 클래스가 정의되어 있음
  - const char * what() 이라는 가상 메쏘드가 정의되어 있음
  - 그 외에도 많은 예외 데이터 형을 정의함, (bad_exception, 등)
- stdexcept 헤더 파일에 logic_error 클래스와 runtime_error 클래스가 정의되어 있음

logic_error 패밀리

- 프로그래밍이 수정될 여지가 있음
- `domain_error` 클래스 → 정의되지 않은 정의역 (log, sqrt, arcsin 등)
- `invalid_argument` 클래스 → 기대하지 않는 값이 함수에 전달되었을 때
- `length_error` 클래스 → 원하는 액션을 취할만큼 충분한 공간을 사용할 수 없을 때 (string 의 append())
- `out_of_bounds` 클래스 → 인덱싱 에러 (`operator[]()`)

runtime_error 패밀리

- 프로그래밍 수정 중에도 나올 수 있으나, 완성된 프로그램으로도 피할 수 없는 곤란한 문제일 수 있음
- `range_error` 클래스 → 함수의 적절한 치역을 벗어날 때
- `overflow_error` 클래스 → 부동 소수점 계산에서 컴퓨터가 나타내는 최고 크기보다 더 큰 수가 계산될 때
- `underflow_error` 클래스 → 부동 소수점 계산에서 0보다 크고 컴퓨터가 나타내는 최소 크기보다 더 작은 수가 계산될 때

처리 예

```
try { ... }
catch (out_of_bounds & oe)
{ ... }  // out_of_bounds 에러 포착
catch (logic_error & oe)
{ ... }  // 나머지 logic_error 에러 패밀리 포착
catch (exception & oe)
{ ... }  // runtime_error, exception 객체들 포착
```

bad_alloc 예외와 new

- new를 사용할 때 일어나는 메모리 할당 문제를 해결하는 두가지 방법 (1051쪽 `newexcp.cpp`)
  - 널 포인터를 리턴
  - bad_alloc 예외를 발생 (`#include <new>`)
    - what() 메쏘드가 적절한 문자열을 리턴함

예외, 클래스, 상속

- 1053쪽의 sales.h
- 하나의 예외 클래스를 다른 클래스로부터 파생시킬 수 있음
  - `class bad_index : public std::logic_error`
- 클래스 정의 안에 예외 클래스 선언을 내포시켜 예외들을 클래스 안에 병합시킬 수 있음
  - `class Sales` 내부의 `class bad_index`
- 내포된 선언들은 상속될 수 있으며, 그들 자신이 기초 클래스 역할을 함
  - `class nbad_index : public Sales::bad_index`

sales.h의 예외 지정

- `virtual double operator[](int i) const throw (std::logic_error) // Sales 버전`
  `virtual double operator[](int i) const throw (std::logic_error) // LabeledSales 버전`
- `std::logic_error`에 `bad_index` 형과 `nbad_index` 형 둘다 매치됨 → 기초 클래스를 던진다고 선언하고 파생 클래스를 던질 수 있음
- 파생 메쏘드는 기초 메쏘드와 동일한 예외 지정을 가지거나 기초 메쏘드에서 사용된 데이터 형으로부터 파생된 데이터 형이어야 하므로, 원칙적으로 Sales 버전은 `bad_index`를 사용하고 LabeledSales 버전은 `nbad_index` 형을 사용할 수 있어야 함

잘못된 예외

- 발생한 예외는 예외 지정자 리스트의 데이터 형들 중 하나와 일치해야 함
- 일치하지 않으면 기대하지 않은 예외(unexpected exception)로 간주되고, 프로그램 실행이 중지
- 예외가 포착되지 (caught) 않으면 포착되지 않은 예외(uncaught exception)로 간주되고, 프로그램 실행이 중지

포착되지 않은 예외

- 기본적으로 exception 헤더 파일 내의 terminate() 함수를 호출 → abort() 함수를 호출
- terminate() 함수의 행동을 바꾸려면...

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();
void terminate();
```

- `set_terminate`
  - 함수 포인터
  - 전달 인자 없고 void가 리턴 형인 함수 포인터를 전달 인자로 받음 (새로 등록할 함수 포인터)
  - 전달 인자 없고 void가 리턴 형인 함수 포인터를 반환함 (이전에 등록된 함수 포인터)

포착되지 않은 예외

상태 메시지 출력하고 exit code 5로 종료 (exit(5))하려면...

1. 헤더 파일 선언

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

2. 예외 처리 함수 정의

```
void myQuit()
{
  cout << "포착되지 않은 예외가 발생하여 프로그램 중지";
  exit(5);
}
```

3. 예외 처리 함수 등록

```
set_terminate(myQuit);
```

기대하지 않은 예외

- 함수 A가 특정 객체 C 예외를 발생시키는 함수 B를 포함한다면, B 뿐만 아니라 A의 예외 지정에도 C가 들어가야 함
- 기본적으로 exception 헤더 파일 내의 unexpected() 함수를 호출 → terminate() 함수를 호출 → abort() 함수를 호출
- unexpected() 함수의 행동을 바꾸려면 set_unexpected 함수 사용

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();
void unexpected();
```

unexpected_handler 함수의 선택 사항

1. terminate() (디폴트 행동), abort() 또는 exit() 호출하여 프로그램 종료
2. 예외 발생 — `unexpected_handler` 대체 함수가 발생시키는 예외와 기대하지 않은 예외를 발생시킨 그 함수의 오리지널 예외 지정에 따라 다르게 행동함
   - 새로 발생한 예외가 오리지널 예외 지정과 일치하면, 그것에 맞는 catch 블럭을 찾아서 진행
   - 새로 발생한 예외가 오리지널 예외 지정과 불일치
     - `std::bad_exception`이 오리지널 예외 지정에 들어 있으면, 그 일치하지 않는 예외는 `std::exception`로 대체됨
     - `std::bad_exception`이 오리지널 예외 지정에 들어 있지 않으면, terminate() 호출

모든 예외를 포착하고 싶다면

```
#include <exception>
using namespace std;
void myUnexpected() {
  throw std::bad_exception(); /* or throw; */
}
set_unexpected(myUnexpected);
double Argh(double, double)
throw(out_of_bounds,bad_exception);
...
try { x=Argh(a,b); }
catch (out_of_bounds & ex) { ... }
catch (bad_exception & ex) { ... }
```

예외 주의사항

- 예외를 사용하면 프로그램 크기가 커지고 실행 속도가 떨어짐
- 예외 지정은 템플릿과 어울리지 않음 — 템플릿 함수들은 특수화에 따라 다른 종류의 예외를 발생시킴
- 동적 메모리 할당과도 안맞음 — 포인터에 의한 동적 메모리 할당의 경우 파괴자 호출이 안됨
  - 예외 안에서 다시 메모리 해제를 함
  - `auto_ptr` 템플릿 사용 (1106쪽)

RTTI의 목적

- Runtime Type Identification
- 프로그램이 실행 도중 객체들의 데이터 형을 식별하는 표준적인 메커니즘을 제공
- 가상 함수를 가지고 있는 클래스들에 대해서만 사용 가능함
- Java의 경우, `java.lang.Class`, `java.lang.reflect.*`

RTTI가 필요한 경우

- 이런 경우 데이터 형을 어떻게 알 수 있나?
  - 업캐스팅되었을 경우
  - 특정 기초 클래스 밑에 여러 파생 클래스가 있을 때 (Command Pattern), 어떤 주어진 정보에 따라 객체를 만들고 그 객체를 기초 클래스 포인터나 참조로 넣고 반환 (Abstract Factory Pattern)
- 데이터 형을 알아야 할 경우
  - 파생 객체가 상속되지 않은 메쏘드를 가지는 경우
  - 디버깅 목적

RTTI를 지원하는 세 가지 요소

- dynamic_cast 연산자 - 기초 클래스 형 포인터로부터 파생 클래스 형 포인터 생성을 시도해 봄

RTTI를 지원하는 세 가지 요소

- dynamic_cast 연산자 - 기초 클래스 형 포인터로부터 파생 클래스 형 포인터 생성을 시도해 봄
- typeid 연산자 - 어떤 객체의 정확한 데이터 형을 식별하는 하나의 값을 반환

RTTI를 지원하는 세 가지 요소

- dynamic_cast 연산자 - 기초 클래스 형 포인터로부터 파생 클래스 형 포인터 생성을 시도해 봄
- typeid 연산자 - 어떤 객체의 정확한 데이터 형을 식별하는 하나의 값을 반환
- type_info 구조체 - 어떤 특별한 데이터 형에 대한 정보를 저장

dynamic_cast 연산자

- 객체 형을 알려준다기 보다는 특정 형의 포인터를 안전하게 대입할 수 있는지 알려줌 (다운캐스팅 가능 여부를 알려줌)

```
class A {};  class B : public A {};
class C : public B {};

A* pa = new A;  A* pb = new B;  A* pc = new C;

C* p1 = (C*) pc; // 안전함
C* p2 = (C*) pb; // 안전하지 않음
B* p3 = (C*) pc; // 안전함
```

dynamic_cast 연산자

- 데이터 형 변환이 안전한지 물어봄으로써 특정 메쏘드를 호출 가능한지 알아냄
  - `B *pb = dynamic_cast<B *>(pa)`
- 안전하게 변환되면 주소를 반환, 아니면 널 포인터인 0을 반환
- 안전한 변환은 업캐스팅을 의미 — *pa가 *pb 객체 클래스와 같은 클래스거나 파생 클래스 객체
- 1069쪽 `rtti1.cpp`
  - `if (ps=dynamic_cast<Superb *>(pg)) ps->say();`

dynamic_cast 연산자

- 참조와 함께 사용할 때는 널 포인터에 해당하는 참조가 없으므로 bad_cast 예외(#include <typeinfo>)를 발생

```
#include <typeinfo>
...
try
{
  Superb & rs = dynamic_cast<Superb &>(rg);
}
catch (bad_cast & e) { ... }
```

typeid 연산자와 type_info 클래스

- typeid 연산자로 두 객체의 데이터 형이 같은지 결정 — 클래스의 이름이나 객체로 평가되는 식을 전달인자로 받음
  - `typeid(Magnificient) == typeid(*pg)`
  - 맞으면 true, 틀리면 false, pg가 널 포인터이면 bad_typeid 예외(`#include <typeinfo>`) 발생
  - 객체 클래스 이름 출력 — `cout << typeid(*pg).name();`
- dynamic_cast와 가상 함수가 처리하지 못하는 상황을 위해 사용함

RTTI의 문제점

- RTTI는 Java나 다른 언어에 비해 미약한 기능
  - http://www.rcs.hu/Articles/RTTI_Part1.htm
- 비판적인 사람들은 프로그램의 효율을 떨어뜨리고, 잘못된 프로그래밍 습관을 초래한다고 주장함
- 가능한 모든 클래스를 명시적으로 비교하는 것은 비효율적이며 객체지향적이 아님 (1077쪽)
- typeid 보다는 dynamic_cast와 가상 함수, 그보다는 가상 함수와 객체 지향 설계로 해결할 수 있는지 고민해야 함

데이터 형 변환 연산자란?

- 과거의 C++에서 데이터 형 변환의 안전성을 개선한 것. 검색하기도 쉬움
- `dynamic_cast` — 안전한 업캐스팅만 허용 (이미 보았음)
- `const_cast` — const 또는 volatile로 또는 그 반대로 변환
- `static_cast` — 업캐스트와 다운캐스트, 기본 형의 수치 변환
- `reinterpret_cast` — 위험한 데이터 형 변환

C는 데이터 형 변환이 느슨함 - 아래 세 개 다 허용됨

- 주소를 문자열로 변환 — `char * pch = (char *) (&d);`
- 주소를 문자로 변환 — `char ch = char (&d);` (C++에선 허용되지 않음)
- 주소를 포인터로 변환 — `Junk * pj = (Junk *) (&d);`

const_cast

- const 또는 volatile로 또는 그 반대로 변환
- const로 선언된 포인터를 변경하지, 그 변수 자체의 const를 없애지는 못함

```
#include <typeinfo>
High bar;
const High * pbar = &bar;
High *pb = const_cast<High *>(pbar); // const를 없앰 (O)
const Low *pl = const_cast<const Low *>(pbar); // High를 Low로 (X)
```

static_cast

- 업캐스트와 다운캐스트, 기본 형의 수치 변환

```
#include <typeinfo>
High bar;
Low blow;
High * pb = static_cast<High *> (&blow); // 업캐스트
Low * pl = static_cast<Low *> (&bar); // 다운캐스트
Pond * pmer = static_cast<Pond *> (&blow); // 무효
```

- double을 int로, float를 long으로 변환 가능

reinterpret_cast

- 위험한 데이터 형 변환

```
struct dat {short a; short b;};
long value = 0xAABB1234;
dat * pd = reinterpret_cast<dat *>(&value);
cout << pd-> a; // 처음 2 바이트 출력
```

- 이식성이 없음. 위의 경우 Big Endian/Little Endian에 따라 다르게 작동함
- 포인터 형을 보다 작은 정수형 또는 부동소수점 형으로 변환 불가능
- 함수 포인터를 데이터 포인터, 또는 그 반대로 바꾸지 못함