**항목 27. 캐스팅은 절약, 또 절약! 잊지 말자.**

캐스팅 문법 정리.

1.     C 스타일의 캐스트.

(T) 표현식 // 표현식 부분을 T 타입으로 캐스팅합니다.

2.     함수 방식 캐스트입니다. 문법이 함수 호출문 같지요.

T (표현식) // 표현식 부분을 T 타입으로 캐스팅합니다.

어떻게 쓰든 이들이 가진 의미는 똑같습니다. 단지 괄호를 어디에 썼느냐 만 다를 뿐이지요.

C++는 4가지로 이루어진, 새로운 형태의 캐스트 연산자를 독자적으로 제공합니다  
(**신형 스타일의 캐스트** 혹은 **C++ 스타일의 캐스트**라고 부르죠)

**const\_cast<**T**>(**표현식**)**

**dynamic\_cast<**T**>(**표현식**)**

**reinterpret\_cast<**T**>(**표현식**)**

**static\_cast<**T**>(**표현식**)**

**  const\_cast**

객체의 상수성(constness)을 없애는 용도 혹은 휘발성(volatileness)을 제거하는 용도로 사용됩니다. 이런 기능을 가진 C++ 스타일의 캐스트는 이것밖에 없습니다.

**  dynamic\_cast**

이른바 ‘안전한 다운캐스팅(safe downcasting)’을 할 때 사용하는 연산자입니다.   
즉, 주어진 객체가 어떤 클래스 상속 계통에 속한 특정 타입인지 아닌지를 결정하는 작업에 쓰입니다.   
구형 스타일의 캐스트 문법으로는 흉내조차도 낼 수 없는 유일한 캐스트이기도 합니다. 덤으로, 신경 쓰일 정도로 **런타임 비용이 높은 캐스트 연산자로도 유일**하고요.

**  reinterpret\_cast**

포인터를 int로 바꾸는 등의 하부 수준 캐스팅을 위한 만들어진 연산자로서, 이것의 적용 결과는 구현환경에 의존적입니다 (이식 성이 없다는 뜻이죠)  
**이런 캐스트는** **하부 수준 코드 외에는 거의 없어야 합니다**.

**  static\_cast**

**암시적 변환[비상수 객체를 상수 객체로 바꾸거나(3), int를 double로 바꾸는 등의 변환]을 강제로 진행할 때 사용**합니다.  
흔히들 이루어지는 타입변환을 거꾸로 수행하는 용도(void\*를 일반 타입의 포인터로 바꾸거나, 기본 클래스의 포인터를 파생 클래스의 포인터로 바꾸는 등)로도 쓰입니다.  
물론 상수 객체를 비 상수 객체로 캐스팅하는데 이것을 쓸 수는 없습니다 (위에서 말한 const\_cast 연산자밖에 안 됩니다).

구형 스타일의 캐스트는 요즘도 적법하게 쓰일 수 있지만, 그보다는 C++ 스타일의 캐스트를 쓰는 것이 바람직합니다.  
우선, 코드를 읽을 때 알아보기 쉽기 때문에, 소스 코드의 어디에서 C++의 타입 시스템이 망가졌는지 찾아보는 작업이 편해집니다.  
둘째, 캐스트를 사용한 목적을 더 좁혀서 지정하기 때문에, 컴파일러 쪽에서 사용 에러를 진단할 수 있습니다. 즉, 상수성을 없애려고 한 부분에다가 const\_cast 대신에 다른 신형 스타일의 캐스트를 실수로 썼다면 코드 자체가 컴파일 되지 않으므로 좋다는 것입니다.

필자가 구형 스타일의 캐스트를 쓰는 경우는 딱 한군데 같아요.  
객체를 인자로 받는 함수에 객체를 넘기기 위해 명시호출 생성자를 호출하고 싶을 경우인데.

|  |
| --- |
| class Widget {  public:             explicit Widget(int size);             ...  };    void doSomething(const Widget& w);    doSomething(**Widget(15)**); // 함수 방식 캐스트 문법으로 int로부터 Widget을 생성합니다.    doSomething(**static\_cast<Widget>(15)**); // C++ 방식 캐스트를 써서 int로부터 Widget을 생성합니다. |

캐스팅은 그냥 어떤 타입을 다른 타입으로 처리하라고 컴파일러에게 알려 주는 것밖에 없다고 생각할 수 있는데, 오해입니다.   
어떻게 쓰더라도(캐스팅으로 명시적으로 바꾸거나, 컴파일러가 암시적으로 바꾸거나) 일단 타입 변환이 있으면, 이로 말미암아 런타임에 실행되는 코드가 만들어지는 경우가 정말 적지 않습니다.

|  |
| --- |
| int x, y;  ...  double d = static\_cast<double>(x)/y; // x를 y로 나눕니다. 그러나 이때 부동소수점 나눗셈을 사용합니다. |

Int 타입의 x를 double 타입으로 캐스팅한 부분에서 코드가 만들어집니다.   
그것도 거의 항상 그렇습니다.   
왜냐하면 대부분의 컴퓨터 아키텍처에서 int의 표현구조와 double의 표현구조가 아예 다르기 때문입니다.

|  |
| --- |
| class Base { ... };  class Derived: public Base { ... };  Derived d;  Base \*pb = &d; // Derived\* => Base\*의 암시적 변환이 이루어집니다. |

파생 클래스 객체에 대한 기본 클래스 포인터를 만드는(초기화하는), 흔한 코드입니다.  
그런데 두 포인터의 값이 같지 않을 때도 가끔 있다는 사실, 아십니까?  
이런 경우가 되면, 포인터의 변위(offset)를 Derived\* 포인터에 적용하여 실제의 Base\* 포인터 값을 구하는 동작이 바로 **런타임(runtime)**에 이루어집니다.

객체 하나(이를테면 Derived 타입의 객체)가 가질 수 있는 주소가 오직 한 개가 아니라,   
그 이상이 될 수 있음을 (Base\* 포인터로 가리킬 때의 주소, Derived\* 포인터로 가리킬 때의 주소) 볼 수 있습니다.  
이런 현상은 C, 자바, C# 에서는 결코 생길 수 없지만 C++에서는 생깁니다.  
사실 C++에서는 다중 상속이 사용되면 이런 현상이 항상 생기지만, 심지어 단일 상속인데도 이렇게 되는 경우가 있습니다.

객체의 메모리 배치구조를 결정하는 방법과 객체의 주소를 계산하는 방법은 컴파일러마다 천차만별입니다. 즉, 어떤 플랫폼에서 ‘메모리 배치를 다 꿰고 있어서’ 캐스팅을 했을 때 문제가 없었을지라도, 다른 플랫폼에서 그게 또 통하지는 않는다는 이야기죠.

가상 함수를 파생 클래스에서 재정의해서 구현할 때, 기본 클래스의 버전을 호출하는 문장을 가장 먼저 넣어달라는 요구사항을 보게 됩니다.

|  |
| --- |
| class Window { // 기본클래스  public:             virtual void onResize() {...} // 기본클래스의 onResize구현결과             ...  };    class SpecialWindow: public Window { // 파생클래스  public:             virtual void onResize() { // 파생클래스의 onResize 구현결과  **static\_cast<Window>(\*this)**.onResize(); // \*this를 Window로 캐스팅하고 그것에 대해 onResize를 호출합니다. 동작이 안됩니다.                       ... // SpecialWindow에서만 필요한 작업을 여기서 수행합니다.             }             ...  }; |

캐스트 부분은 \*this를 Window로 캐스팅하는 코드다.   
이에 따라 호출되는 onResize()는 Window::onResize 가 됩니다.   
그런데, 함수 호출이 이루어지는 객체가, 현재의 객체가 아닙니다.

이 코드에서는 캐스팅이 일어나면서 \*this의 기본 클래스 부분에 대한 **사본**이 임시적으로 만들어지게 되어 있는데, 지금의 onResize는 바로 이 임시 객체에서 호출된 것입니다.

이 문제를 풀려면 일단 캐스팅을 빼버려야 합니다.

|  |
| --- |
| class SpecialWindow: public Window { // 파생클래스  public:             virtual void onResize() { // 파생클래스의 onResize 구현결과  **Window::onResize();** // \*this에서                       ... // Window::onResize를 호출합니다.             }             ...  }; |

dynamic\_cast, 말도 많고 탈도 많은 연산자입니다.   
지금은 상당수의 구현환경에서 이 연산자가 정말 느리게 구현되어 있다.

dynamic\_cast 연산자를 쓰고 싶어지는 때가 있다.   
파생 클래스 객체임이 분명한 녀석이 있어서, 이에 대해 파생 클래스의 함수를 호출하고 싶은데,   
그 객체를 조작할 수 있는 수단으로 기본 클래스의 포인터 (혹은 참조자) 밖에 없을 경우는 적지 않게 생기거든요. 이런 문제를 피해가는 일반적인 방법으로는 2가지를 들 수 있습니다.

첫번째 방법은, 파생 클래스 객체에 대한 포인터(혹은 스마트 포인터(13))를 컨테이너에 담아둠으로써, 각 객체를 기본 클래스 인터페이스를 통해 조작할 필요를 아예 없애 버리는 것입니다.   
아래처럼 하지 말고

|  |
| --- |
| class Window {...};  class SpecialWindow: public Window {  public:             void blink();             ...  };    typedef std::vector<std::tr1::shared\_ptr<Window> > VPW; // tr1::shared\_ptr (13)  VPW winPtrs;  ...  // 그다지 바람직스럽지 않은 코드: dynamic\_cast를 쓰고 있습니다.  for (VPW::iterator iter = winPtrs.begin();             iter != winPtrs.end();             ++iter) {             if (SpecialWindow \*psw = **dynamic\_cast**<SpecialWindow\*>(iter->get()))                       psw->blink();  } |

다음과 같이 해 보라는 거죠.

|  |
| --- |
| typedef std::vector<std::tr1::shared\_ptr<**SpecialWindow**> > **VPSW**;  **VPSW** winPtrs;  ...  // 더 괜찮은 코드: dynamic\_cast가 없습니다.  for (VPSW::iterator iter = winPtrs.begin();             iter != winPtrs.end();             ++iter) {  **(\***iter**)**->blink();  } |

이 방법으로는 Window에서 파생될 수 있는, 모든 포인터를 똑같은 컨테이너에 저장할 수는 없습니다.

한편, Window 에서 뻗어 나온 자손들을, 전부 기본 클래스 인터페이스를 통해 조작할 수 있는 다른 방법이 없는 것은 아닙니다. 여러분이 원하는 조작을 가상 함수 집합으로 정리해서 기본 클래스에 넣어두면 됩니다.   
예를 들어, 지금은 blank()가 SpecialWindow에서만 가능하지만, 아무것도 안하는 기본 blink()를 구현해서 가상합수로 제공하는 것입니다.

|  |
| --- |
| class Window {  public:             virtual void blink() {...} // 기본구현은 '아무 동작 안하기' 입니다.             ...  };    class SpecialWindow: public Window {  public:             virtual void blink() {...} // 이 클래스에서는 blink가 특정한 동작을 수행합니다.                       ...             }             ...  };    typedef std::vector<std::tr1::shared\_ptr<Window> > VPW;  VPW winPtrs; // 이 컨테이너는 Window에서 파생된 모든 타입의 객체에 대한 포인터들을 담습니다.  …    for (VPW::iterator iter = winPtrs.begin();             iter != winPtrs.end();             ++iter) {             (\*iter)->blink(); // dynamic\_cast가 없습니다. |

위 2가지 방법중 어떤 것도(타입 안전성을 갖춘 컨테이너를 쓰든지, 가상함수를 기본클래스 쪽에 올려두든지) 모든 상황에 다 적용하기란 불가능하지만, 상당히 많은 상황에서 dynamic\_cast를 쓰는 방법 대신에 꽤 잘 쓸 수 있습니다.

정말 피해야 하는 설계가 있습니다.   
바로 폭포식(cascading) dynamic\_cast라고 불리는 구조인데,

|  |
| --- |
| class Window {...};  ... // 파생 클래스가 여기서 정의됩니다.  typedef std::vector<std::tr1::shared\_ptr<Window> > VPW;  VPW winPtrs;  ...  for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end(); ++iter)  {             if (SpecialWindow1 \*psw1 = **dynamic\_cast**<SpecialWindow1\*>(iter->get())) {...}             else if (SpecialWindow2 \*psw2 = **dynamic\_cast**<SpecialWindow2\*>(iter->get())) {...}             else if (SpecialWindow3 \*psw3 = **dynamic\_cast**<SpecialWindow3\*>(iter->get())) {...}  ...  } |

이런 C++ 코드 때문에 C++가 욕을 먹는 것입니다.  
크기만 하고 아름답지 않으며, 속도도 둔한데다가, 망가지기 쉬운 코드가 만들어지거든요.  
Window 클래스 계통이 바뀌었다는 소식이라도 들렸다 치면, 항상 이런 코드는 ‘또 뭐 넣고 뺄거 없나?’ 하는 검토 대상이 되니까 말이죠 (파생 클래스가 하나 추가되면, 위의 폭포식 코드에 계속해서 조건 분기문에 우겨 넣어야 합니다).  
이런 코드는 가상 함수 호출에 기반을 둔 어떤 방법이든 써서 바꿔 놓아야 합니다.

캐스팅 역시, 그냥 막 쓰기에는 꺼림직한 문법 기능을 써야 할 때, 흔히 쓰이는 수단을 활용해서 처리하는 것이 좋습니다. 쉽게 말해 최대한 격리시키라는 것입니다. 캐스팅을 해야 하는 코드를 내부 함수 속에 몰아 놓고, 그 안에서 일어나는 ‘천한’일들을 이 함수를 호출하는 외부에서 알 수 없도록 인터페이스로 막아두는 식으로 하면 됩니다.

**요약**.

\* 다른 방법이 가능하다면 캐스팅은 피하십시오. 특히 수행 성능에 민감한 코드에서 dynamic\_cast는 몇 번이고 다시 생각하십시오. 설계 중에 캐스팅이 필요해졌다면, 캐스팅을 쓰지 않는 다른 방법을 시도해 보십시오.

\* 캐스팅이 어쩔 수 없이 필요하다면, 함수 안에 숨길 수 있도록 해 보십시오. 이렇게 하면 최소한 사용자는 자신의 코드에 캐스팅을 넣지 않고 이 함수를 호출할 수 있게 됩니다.

\* 구형 스타일의 캐스트를 쓰려거든 C++ 스타일의 캐스트를 선호하십시오. 발견하기도 쉽고, 설계자가 어떤 역할을 의도 했는지가 더 자세하게 드러납니다.