More Effective C++

# 동기

C++언어 도사가 되어라.

# 목적

C++을 효과적으로 사용한다.

설계 기술을 향상시킨다.

# 과정

항목별로 중요한 내용을 정리하고 나아가 깊이 있는 통찰을 보여준다.

# [항목1] 포인터와 참조자를 구분하자.

포인터(\*)와 참조자(&)가 뭔지 아는 것은 어렵지 않다. 둘 다 어떤 객체의 주소를 취해서 접근한다.

이제부터는 각각의 사용을 어떻게 구분해야 좋을지 고려해볼 것이다.

## 널 참조자는 없다.

참조자는 반드시 어떤 값을 참조하고 있어야 한다. 반면, 포인터는 널 값을 가리킬 수 있다.

### 이게 뭐 어쨌다는 것인데?

새로운 제약이 있으므로 포인터가 더 좋아 보이는가? 그렇다면 프로그래밍 왕초보다.

void attackTarget(Archer\* target)

{

int myGreatPower = 9999;

// ...

// 참조하는 대상이 사라지지 않았는지 검사해야 한다.

if (target)

{

target->hit(myGreatPower);

}

}

void attackTarget(Archer& target)

{

int myGreatPower = 9999;

// ...

// 참조자는 검사할 필요가 없다.

target.hit(myGreatPower);

}

코드가 간결해졌고, 예상치 못한 참조 대상 소멸현상을 피할 수 있다. **그러므로 널 값을 가리켜서는 안 되는 변수라면 참조자를 쓰는 것이 현명하다.**

## “다른 객체를 참조하게 할 수 있는가”의 여부

참조자는 한번 참조하면 참조 대상을 바꿀 수 없지만, 포인터는 바꿀 수 있다.

int main()

{

Archer dk("dk");

Archer yth("yth");

auto& champion = dk;

auto loser = &yth;

champion = yth; // 참조에 대입을 하면 주소가 바뀌지 않고 값이 바뀐다.

loser = &dk;

std::cout << "who is champion?\n";

champion.intro(); // output: yth

std::cout << "who is loser?\n";

loser->intro(); // output: yth

dk.intro(); // output: yth

yth.intro(); // output: yth

return 0;

}

흔하디 흔한 기초 테스트지만 배울 점이 있다. 참조를 하는 변수는 주소를 바꿀 수 없다. 그럼 참조자에 대입을 하면 어떻게 될까? 컴파일 에러를 발생시킬까? 대입을 하면 그 참조된 주소의 값을 바꿔버린다. 그래서 위의 결과에서 두 개의 변수가 같아지는 모습을 볼 수 있다. 이는 포인터와 확실하게 다르다. 포인터는 참조하는 주소 자체를 바꾸기 때문이다.

**만약 어떤 객체의 참조가 바뀔 일이 없다는 점이 보장된다면, 참조를 사용하는 것이 안전하고 현명하다.**

## 연산자 함수 구현

둘은 다르게 생겼기 때문에 구현 시, 인터페이스도 다를 것이다. 그 중 참조자를 반드시 써야 하는 특수한 상황이 있는데, 연산자 함수를 구현할 때이다.

reference operator[](size\_type \_Pos) const

{

// ...

return (\*(this->\_Myfirst() + \_Pos));

}

어떤 타입에 대한 참조를 반환하는 STL의 구현이다. 만약 포인터를 사용하게 된다면,

template<typename T>

class MyVector

{

public:

MyVector(int size) : arr(new T[size]), size(size)

{

for (int i = 0; i < size; i++)

{

\*(arr + i) = 0;

}

}

~MyVector()

{

delete[] arr;

}

T& operator[](int size)

{

return (\*(arr + size));

}

/\*T\* operator[](int size)

{

return (arr + size);

}\*/

void printValue()

{

for (int i = 0; i < size; i++)

{

std::cout << \*(arr + i) << " ";

}

std::cout << "\n";

}

private:

T\* arr;

int size;

};

int main()

{

MyVector<int> v(10);

v[5] = 9999;

//\*v[5] = 9999;

v.printValue();

return 0;

}

주석 부분이 포인터를 반환하는 연산자 함수이다. 포인터를 사용하게 되면 어색한 부분이 있다. 값을 찾기 위해서 \*v[i]형태로 작성해야 한다. 그러면 마치 v는 포인터의 배열로 착각하게 만들 수 있으므로 혼동을 유발한다. **그러므로 참조자를 반환하게 되면 실제 객체를 반환한 것처럼 사용하므로 깔끔하게 구현할 수 있다.**

## 나머지의 경우

위의 3가지는 참조자와 포인터와 다른 점을 살펴본 것이다. 이러한 경우에는 참조자를 사용하는 것이 확실히 효율적이다.

하지만 분명히 포인터와 참조자는 다르다. 포인터는 주소 자체를 나타내므로 좀 더 원시적인 코드를 작성할 때 반드시 필요하다. **그러므로 위의 3가지의 경우가 아니라면 포인터를 사용하라.**

# [항목2] 가능한 C++스타일의 캐스트를 즐겨 쓰자.

C++스타일의 캐스트는 4가지가 있다.

## C++스타일 캐스트의 정의

**const\_cast<T>:** 객체의 const(상수성) 혹은 휘발성(volatile)을 없애줍니다.

**dynamic\_cast<T>:** 안전한 다운캐스팅(어떤 객체가 상속 계통에 속한 특정 타입인자 아닌지를 결정할 때)에 쓰입니다.

**reinterpret\_cast<T>:** 포인터를 int로 바꾸는 등의 하부 수준의 캐스팅에 쓰입니다. C스타일의 캐스팅과 가장 유사합니다.

**static\_cast<T>:** 암시적 변환을 강제로 진행할 때 사용합니다.

구질구질한 구형 캐스트보다는 신형 캐스트를 선호합니다. 이유는 아래와 같습니다.  
1. 타입 시스템이 어디서 망가졌는지 찾아보기 쉽습니다.  
2. 컴파일러쪽에서 사용 에러를 진단할 수 있습니다.

## C++스타일 캐스트를 사용해보자.

### const\_cast<T>

class Archer

{

public:

void showLevel() const

{

std::cout << "level up: " << level << "\n";

}

void levelUp()

{

const\_cast<int&>(level) += 5;

showLevel();

}

private:

const int level{ 1 };

};

int main()

{

const int&& myExperience = 9999;

// myExperience += 5;

// <>안에는 포인터나 참조값이 들어가야 한다. 아래는 컴파일이 안 된다.

// auto var = const\_cast<int>(myExperience);

auto& var = const\_cast<int&>(myExperience);

var += 5;

std::cout << myExperience << "\n"; // 10004

Archer a;

a.levelUp();

return 0;

}

참고로 const\_cast는 형식 한정자만 조절할 수 있다. int대신 double을 넣는다면 컴파일 에러가 발생한다.

### static\_cast<T>

// 한정자도 넣을 수 있다. 하지만 참조나 포인터는 넣을 수 없다.

auto var = static\_cast<const double>(myExperience);

DarkKnight dk;

auto newA = static\_cast<Archer>(dk);

암시적 변환을 강제한다. 바뀔 수 있다면 수행하는 안전한 캐스팅이다.

### dynamic\_cast<T>

// 이런 것은 존재할 수 없다.

// auto var = dynamic\_cast<const double>(myExperience);

// <>안에는 포인터나 참조 형식이어야 한다.

// auto newA = dynamic\_cast<Archer>(dk);

auto greatArcher = new DarkKnight;

auto newA = dynamic\_cast<Archer\*>(greatArcher);

auto newDK = dynamic\_cast<DarkKnight\*>(newA);

얼핏 보면 static\_cast와 비슷하지만, 분명히 다르다. dynamic\_cast는 반드시 파생 관계에서 캐스팅을 하기 때문에 포인터나 참조 형태로 되어 있어야 한다. 게다가 static\_cast은 자식->부모로만 변환 가능하지만, dynamic\_cast는 다운 캐스팅(부모->자식)도 가능하게 함으로써, 해당 부모가 임의의 자식으로 변환 가능한지 아닌지를 파악하려면 이 캐스트 연산자만이 가능하다.

// 참고로 기본->파생으로 변환 시에는 기본에 다형성이 보장되어야 합니다.

// 그러므로 virtual 함수가 있어야 합니다.

class Archer

{

public:

virtual void showLevel() const

{

std::cout << "level up: " << level << "\n";

}

void levelUp();

private:

const int level{ 1 };

};

if (dynamic\_cast<DarkKnight\*>(dynamicA))

{

std::cout << "dark knight!\n";

}

if (dynamic\_cast<ArchMage\*>(dynamicA))

{

std::cout << "arch mage!\n"; // 출력 안 됨

}

하지만 성능에 문제가 많으므로 사용을 자제해야 한다.

### reinterpret\_cast<T>

비트를 다시 해석하는 연산자이다. 이는 컴파일러마다 구현이 다르므로 이식성이 없다.

int power = 9999;

// auto reinVar = reinterpret\_cast<Archer>(power); // 이런 터무니 없는 것은 안 된다.

auto reinVar = reinterpret\_cast<double&>(power);

auto reinVar2 = reinterpret\_cast<double\*>(power);

std::cout << reinVar << "\n";

std::cout << reinVar2 << "\n";

//std::cout << \*reinVar2 << "\n"; // 안 됨

auto rein\_greatArcher = new DarkKnight;

auto rein\_dynamicA = reinterpret\_cast<Archer\*>(greatArcher);

auto rein\_dynamicDK = reinterpret\_cast<DarkKnight\*>(dynamicA);

역시 <>안에는 참조나 포인터 형태가 들어가야 하는 것으로 보인다. 보는 것처럼 거의 만능으로 위의 캐스팅 연산을 처리할 수 있다. 동시에 위험한 캐스팅 연산자이다.

// pointer to function to another and back

void(\*fp1)() = reinterpret\_cast<void(\*)()>(f);

// fp1(); undefined behavior

int(\*fp2)() = reinterpret\_cast<int(\*)()>(fp1);

std::cout << std::dec << fp2() << '\n'; // safe

특히 함수 포인터 형식을 재해석하는 기능으로 유용하게 사용할 수 있다.

# [항목3] 배열과 다형성은 같은 수준으로 놓고 볼 것이 아니다.

## 주제: 다형성

다형성: 서로 다른 객체가 동일한 메시지에 대하여 서로 다른 방법으로 응답하는 성질

배열을 쉽게 다룰 수 있게 해주는 도구 중 하나가 다형성이다. 실제로 해당 객체가 정확하게 무엇인지 몰라도 각자의 성질에 맞게 행동할 수 있게 해준다.

## 문제 제기: 다형성과 일반 배열

정확하게 모른다고 배열에서 파생 클래스를 기본 클래스로 다루면 서로 크기가 다르기 때문에 엉뚱한 값을 도출한다. 다형성은 같은 메시지에 다르게 반응할 뿐이다. 둘은 다른 객체이다. 그냥 다르다.

중요한 점은, 포인터로 배열을 다룬다면, 기본 클래스와 파생 클래스를 자유자재로 이용할 수 없다는 점이다. 서로 크기가 다르기 때문이다.

## 해결책

없다. 조심하는 수밖에 없다.

# [항목4] 쓸데없는 기본 생성자는 그냥 두지 말자.

## 주제: 기본 생성자

기본 생성자: 특별히 무엇을 하기는 귀찮은데 객체 하나는 얻고 싶다.

아무런 인자가 없는 생성자이다. 아무런 정보를 주지 않아도 객체를 만든다는 점에서 조금 특별하다. 그러므로 객체를 생성(배열, 컨테이너 등)하는 데에서 편의를 가진다.

## 문제 제기: 기본 생성자의 폐해

하지만 편의에 대한 대가는 따른다. 아무런 초기화 없이 객체를 생성한다는 의미는 모든 정보를 이후에 검사해야 한다는 의미이고, 성능 문제와 코드 비대화 문제가 생긴다.

객체의 생성자는 객체가 온전하게 생성됨을 보장해야 한다. 객체의 목적와 상관없는 행위는 코드를 더럽힌다.

## 해결

### 기본 값 부여

기본 생성자에 값을 부여하는 것이다. 값을 비워두는 것보다는 낫다.

Archer(int level = 1)

{

}

### 펙토리 함수

객체의 생성을 정적 함수가 담당한다. 이는 파생 관계에 있는 객체들을 간단하게 생성하는 코드를 만들 수 있다.

**팩토리 메소드 패턴(Factory Method Pattern) - 객체 생성관련 패턴**  
  
**유용한 경우**

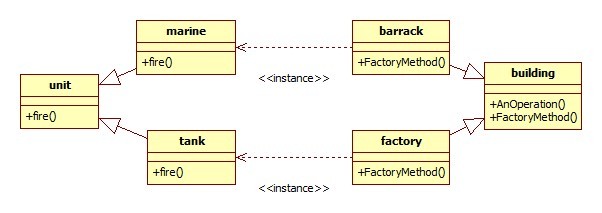
1. 구체적으로 어떤 클래스의 객체를 생성해야 할지 미리 알지 못할 경우 Factory Method Pattern이 유용하다. 구체적으로 생성될 객체는 하위 클래스에 의해 결정이 되며 하위 클래스는 계속 확장이 가능하다.  
  
2. Factory Method Pattern은 하위 클래스들에게 개별 객체의 생성 책임을 분산시켜 객체의 종류별로 객체 생성과 관련된 부분을 구지화 시킬 때 유용하다.

**장점**

1. 어떤 객체를 생성할 것인지와는 무관하게 동일한 형태로 프로그래밍이 가능하다. 이 것은 두개의 상위 클래스가 있기 때문에 가능하다.   
  
2. 직접 생성자를 호출해서 객체를 생성하는 것보다 훨씬 유연하고 확장성 있는 구조이다.

**단점**

1. 생성할 객체의 종류가 달라질때 마다 새로운 하위 클래스를 정의해야 한다. 이는 불필요하게 많은 클래스를 정의해야 한다는 문제점이 있다.



### 초기화 함수

초기화 관련 내용들을 한 함수로 빼놓는다. 여기서 모두 예외를 처리한다. 하지만 어차피 쓸모 없는 검사를 해야 한다는 점에서 의미가 희미하다.

### 배열 포기

그냥 배열을 쓰지 않는다.

# [항목5] 사용자 정의 타입변환 함수에 대한 주의를 놓지 말자.

암시적 변환은 매우 편리해서 우리의 손과 발이 되어줄 것 같지만 득보다는 실이 많다. 어떤 면에서 그러한지 알아보고 이를 해결하기 위한 명확한 해결책을 제시한다.

## 주제: 타입변환

타입변환이란, 타입이 바뀌는 것이다. int형이 double형으로 바뀌는 것이 그러한 사례이다. 어떻게 타입을 변환할 수 있을까?

### 캐스팅한다.

당연한 소리다. 앞서 이해한 캐스트는 적절한 경우에 사용자가 원하는 타입으로 변환해준다. 분명 double형의 자료를 int형으로 바꾸었다.

auto archer1 = static\_cast<int>(9999.0);

너무 쉬워서 하품이 날 지경이다.

### 단일 인자 생성자

인자가 1개뿐인 생성자이다. 생성자는 객체를 생성시에 호출되는 함수인데 무슨 타입변환을 한다는 소리일까? 타입변환과는 전혀 상관없어 보인다. 하지만 다음을 보자.

class Archer

{

public:

Archer(int level)

{

}

};

Archer archer2 = 10;

Archer archer3(10);

//Archer archer2 = Archer(10);

archer3 = 9999;

놀랍게도 Archer이 10이란다. 직관적으로 말이 안 된다. 하지만 이를 허용하는 프로그래머의 프로그래밍 실력도 말이 안 되는 것이다.   
여하튼 무슨 일이 생겼는가? 일단 archer1의 대입 연산자는 대입이 아니다. 타입의 선언과 동시에 사용된 대입 연산자는 기본 생성자의 호출과 정확하게 일치한다. 심지어 마지막 줄의 실제로 대입하는 과정도 문제가 없이 대입되었다. 즉, 정수가 Archer타입으로 변환된 것이다.

어떻게 이가 가능한 것인가? 나머지는 그냥 기본 생성자를 호출한 격이기 때문에 사실 archer3만 살펴보면 된다. 여기서 archer3와 9999은 타입이 다른데 대입을 해야 한다. 여기서 컴파일러는 같은 타입을 만들기 위해 무언가를 시도한다. 여기서 Archer클래스에 단일 인자 생성자가 있으면, 이를 9999에 적용시켜 대입하면 가능하므로 타입변환이 생기는 것이다.

### 암시적 타입변환 연산자

기본적으로 언어 차원에서 암시적 변환을 허용하는 경우가 있다. 또한, 다음과 같이 만들 수 있다.

class Archer

{

public:

Archer(int lv) : level(lv);

operator double() const

{

return static\_cast<double>(level);

}

private:

int level{ 1 };

};

int main()

{

double myVar = Archer(9999);

std::cout << myVar << '\n';

return 0;

}

Archer안에 들어있는 double연산자는 해당 값은 반환한다는 의미다. myVar에 값을 집어넣어야 하는데 일단 타입이 같지가 않다. 그러면 컴파일러는 타입을 같게 만들 수 있도록 타입변환이 가능한지를 살펴본다. 그러던 중, operator double()이 떡 하니 있으므로 Archer(9999)는 이를 찾아 적용시켜 아무 말 없이 타입 변환이 생기는 것이다.

## 문제 제기: 암시적 타입 변환의 문제

프로그래머가 의도치 않은 변화가 생긴다.

## 해결

이를 해결할 수 있는 방법이 있다.

### 단일 인자 생성자

단일 인자 생성자는 explicit을 선언하여 암시적 변환을 막는다.

### 암시적 타입변환 연산자

암시적 타입변환 연산자를 정의하지 않는다.

### 프록시 객체

대리자 클래스를 내부에 두는 것이다. 이는 타입변환에 방지하도록 그를 대신하는 객체로 둔다.

class Archer

{

public:

class Level

{

public:

Level(int lv) : value(lv) {}

int get() const { return value; }

private:

int value{ 1 };

};

Archer(Level lv) : level(lv)

{

}

//operator double() const

//{

// return static\_cast<double>(level);

//}

private:

Level level{ 1 };

};

더 이상 암시적 변환 따위는 생기지 않는다. 왜냐하면 사용자 정의 타입변환 함수는 두 개 이상 쓰이지 않는다라는 규칙이 있기 때문이다.

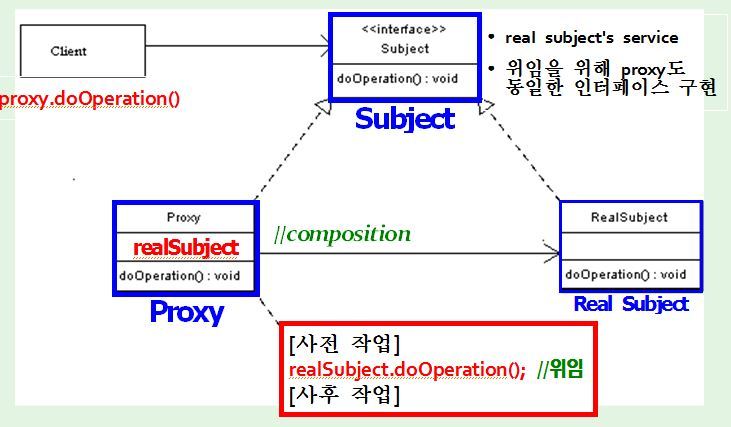
### 프록시 패턴

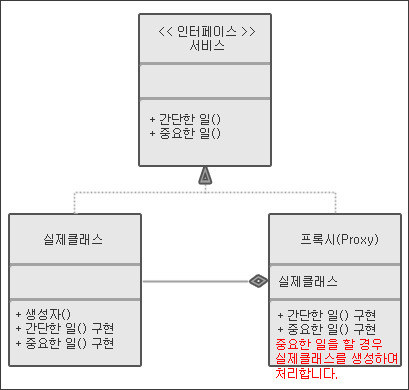
**프록시 패턴이 필요한 이유**

사용 여부가 불확실한 덩치가 큰 객체의 생성은 사용되지 않은 채로 프로그램이 종료되면 시간/자원의 낭비를 초래하기 때문에

**프록시 패턴이란?**

어떤 객체에 대한 접근을 제어하기 위한 용도로 대리인이나 대변인에 해당하는 객체를 제공하는 패턴





class IArcher

{

public:

virtual void intro() = 0;

virtual void powerSkill() = 0;

};

class ArcherProxy : public IArcher

{

public:

virtual void intro() override

{

// 간단한 일은 여기서 처리

}

virtual void powerSkill() override

{

// 사전 처리

// 복잡한 일은 실제 객체에게 넘긴다.

auto real = new Archer;

real->powerSkill();

// 사후 처리

}

};

class Archer : public IArcher

{

public:

Archer() {}

virtual void intro() override

{

// 간단한 일은 여기서 처리

}

virtual void powerSkill() override

{

// 복잡한 일

}

};

# [항목6] 증가 및 감소 연산자의 전위/후위 형태를 반드시 구분하자.

기본적으로 증가의 시점이 다르다는 점을 기억하고 있다. 하지만 이의 내부적인 차이와 이로 인한 구현의 연구가 필요하다.

## 주제: 전위/후위 증가, 감소 연산자

구현은 다음과 같다.

class MyInteger

{

public:

MyInteger(int v) : value(v) {}

MyInteger& operator+=(int add)

{

this->value += add;

return \*this;

}

// prefix

MyInteger& operator++()

{

\*this += 1;

return \*this;

}

// postfix

const MyInteger operator++(int add)

{

const MyInteger oldValue = \*this;

++(\*this);

return oldValue;

}

private:

int value{ 0 };

};

int main()

{

MyInteger me = 9999;

int you = 1;

//you++++; // 안 됨

//++you++; // 역시 안 됨

(++you)++;

++++you;

//me++++; // 안 됨

//++me++; // 역시 안 됨

(++me)++;

++++me;

return 0;

}

## 핵심

변화된 객체를 반환하는 경우는 연산자에 한해야 한다. 객체를 직접 바꿀 수 있는 포인터를 반환한다면 그 객체는 이미 안전하지 못한 것이기 때문이다. 반면, 객체를 반환하는 경우에 변할 일이 없는 객체라면 const로 그것을 명확하게 명시해야 한다. 그러면 실수가 없다.

또 하나는 후위 연산자가 전위 연산자를 이용하여 구현했다는 점이다. 이는 전위 연산자를 수정하면 후위 연산자도 자동으로 수정된다. 둘 간의 연관성이 보존되는 것이다.

# [항목7] &&, ||, 쉼표 연산자는 오버로딩 대상이 절대로 아니다.

왜 이 셋만 가지고 호들갑이냐? 이 셋만 다르기 때문이다.

## 주제: 오버로딩, 단축평가(short-circuit)

int op1() { return 9999; }

int op2() { return 1; }

bool specialCheck()

{

while (true);

}

//typedef int(\*Func)();

using Func = int(\*)();

int ArcherCheck(Func op1, Func op2)

{

if (op1() > op2() && specialCheck())

{

return 1;

}

return 0;

}

int main()

{

std::cout << ArcherCheck(op2, op1) << '\n';

std::cout << ArcherCheck(op1, op2) << '\n'; // 절대 안 끝남

return 0;

}

연산자 &&에서 앞의 표현식에서 결과가 나올 수 있으면 뒤의 표현식을 검사하지 않는다. 이를 단축평가라 한다. 연산자 &&, ||에 대해서는 이를 적용한다.

## 문제 제기: 오버로딩하면 단축평가가 일어나지 않는다.

우리는 기본적으로 위의 연산자에 단축평가가 일어난다고 생각하고 프로그램을 작성하지만, 오버로딩하게 되면 반대인 함수호출 의미구조가 되므로 오류가 생길 여지가 크다.

그리고 원래의 쉼표 연산자는 왼쪽 표현식을 수행하고 오른쪽 표현식을 다음으로 수행한다. 하지만 흉내낼 수 없다. 왜냐하면 어느 표현식이 먼저 수행될 지 보장할 수 없기 때문이다.

그리고 위의 3가지가 중요한 이유는, 오버로딩이 가능하지만 하면 생각처럼 구현되지 않는 연산자들이기 때문이다.

## 해결

위의 3개의 연산자와 오버로딩할 수 없는 연산자들을 기억해둔다. 그리고 그것들은 오버로딩하지 않는다.

# [항목8] new와 delete의 의미를 정확히 구분하고 이해하자.

Operator new란, new연산자가 생성자와 함께 호출하는 메모리 할당하는 연산자다.

#include <cstdio>

#include <cstdlib>

// replacement of a minimal set of functions:

void\* operator new([std::size\_t](http://en.cppreference.com/w/cpp/types/size_t) sz) {

[std::printf](http://en.cppreference.com/w/cpp/io/c/fprintf)("global op new called, size = %zu**\n**",sz);

return [std::malloc](http://en.cppreference.com/w/cpp/memory/c/malloc)(sz);

}

void [operator delete](http://en.cppreference.com/w/cpp/memory/new/operator_delete)(void\* ptr) noexcept

{

[std::puts](http://en.cppreference.com/w/cpp/io/c/puts)("global op delete called");

[std::free](http://en.cppreference.com/w/cpp/memory/c/free)(ptr);

}

int main() {

int\* p1 = new int;

delete p1;

int\* p2 = new int[10]; // guaranteed to call the replacement in C++11

delete[] p2;

}

또한, 실무에서 어떻게 쓰이는지 간단한 자료

프로그램을 개발하신 분들이 공통적으로 한번씩은 격는 장애가 있습니다.

바로 "**메모리 릭**"

new 연산자는 우리에게 실행 시점에의 메모리 할당이 가능하게 하여

필요한 시점에 os가 관리하는 메모리를 사용자 어플리케이션이 동적으로 사용할 수 있게 해주는 아주 고마운 녀석입니다.

하지만 잘못 사용하게 되면, 메모리가 줄줄줄...

거기다가 새는 메모리가 일정이상 넘어가면 OS님하가 "이런! 위험한놈!" 이라고 하시면서 해당 프로세스를 죽여버리십니다.

만약 게임 서버와 같은 복잡한 프로그램에서 서비스 중에 간헐적으로 메모리 릭이 발생하게 되고, 재현이 어렵다면!?!?!?

프로그래머는 "도데체 어디서 새는거야!!" 라면서 머리를 쥐어 뜯으며, 소스를 이잡듯이 뒤지는 상황이 발생합니다.

**타고난 능력자 님들을 제외하고는** 이러한 경험을 한두번씩은 겪으신 분들이 많이 계실거라 생각합니다.

특히나 팀프로젝트 특성상 내가 아니더라고 신입개발자라던지.. 누군가는 만들어 낼 수도 있구요.

(나만 그랬나.................ㅠㅠ)

본 글에서는 메모리 릭 발생 시 어떤놈이 릭을 발생 시키는 주범인지 쉽게 유추가 가능하도록 도와주는 녀석을 만들고자 합니다.

본론으로 들어가서..

c++에서 new연산자의 역할은 무었일까요?

**첫째로**가장 중요한 힙영역에의 실행 시점에서의 메모리 동적 할당이 있을 것이고,

**둘째로**클래스의 경우 생성자(constructor) 의 호출로 객체 초기화를 해 준다는 것

**셋째로**반환되는 주소 영역에 대한 형의 결정

다음 3가지가 있을 것입니다.(또 있나...ㅡㅡ;)

이때 우리가 하고자 하는 new 연산자에 대한 오버로딩은 첫번째 역할인**"메모리 공간에 대한 할당"**만을 재정의 할 수 있습니다.

사용 법은 아래와 같이 매우 간단합니다.

·미리보기 | 소스복사·

1. **void**\* operator **new**(...){
3. }

그럼 이쯤에서 의문이 드실 수도 있을 겁니다.

**"저걸로 메모리 릭을 어떻게 찾는다는 거지?"**

자! 재정의가 가능하다는 것은 new가 호출 될 때 트리거링 되는 루틴을 만들수 있다는 것을 의미합니다.

한단계씩 진행 해 보겠습니다.

먼저 아래와 같이 동적으로 할당된 메모리에 대한 상세 정보를 저장할 struct를 정의 합니다.

(본 글의 예제로는 struct를 사용하지만 입맞에 맞게 사용하시면 될 것 같습니다.)

·미리보기 | 소스복사·

1. **typedef** **struct** tagMemAllocationInfo
2. {
3. // new를 사용한 소스파일 명 편의상 char배열을 사용
4. char   filename[MAX\_FILE\_NAME];
5. **int**    line;         // new를 사용한 소스파일의 라인
6. **time\_t** allocate\_time // 메모리 할당 시간
7. **void**\*  addr;          // 메모리 할당 주소
8. }MEM\_INFO;

그리고 요런식의 맵 타입을 하나 정의해 줍니다.

·미리보기 | 소스복사·

1. // 프로세스의 메모리 동적 할당 된 정보를 저장하는 맵
2. **typedef** std::map<**void** \*, MEM\_INFO>  MEN\_INFO\_MAP;
3. // 이얍! -ㅁ- 일단 개념만 설명하기 위해.. 귀차니즘으로 전역 선언
4. **static** MEN\_INFO\_MAP CurMemAllocationInfoMap;
5. **static**int CurMemAllocationCount = 0;

게임코디 소스편집기에 바로 짜고 있는 관계로.. STL의 사용 및 용법에 대한 부분은 이해 부탁드리구요..;

일단 map을 사용한 이유는 delete호출 시 direct search를 하여 삭제해야 한다는 것을 어필하기 위함입니다.(성능)

물론 위의 맵과 카운팅 하는 부분은 DynamicMemoryManager 클래스로 만드시는게 좋겠지요? (귀차니즘...)

마지막으로 new와 delete연산자를 재정의 해줍니다.

아참 여기서 중요한 것은 첫번째 인자는 무조건 size\_t가 들어가야 한다는 것!

·미리보기 | 소스복사·

1. **void**\* operator **new**(**size\_t** sz, **char**\* filename, **int** line)
2. {
3. MEM\_INFO meminfo;
4. **int** len;
6. **void**\* pfs = malloc(sz);
8. **if**(NULL == pfs)
9. {
10. cerr<<"[크리티컬]힙이 없네용"<<endl;
11. exit(-1);
12. }
14. meminfo.addr = pfs;
15. meminfo.line = line;
16. time(&meminfo.allocate\_time);
18. len = strlen( filename );
19. **if**( len > MAX\_FILE\_NAME )
20. {
21. cerr<<"[크리티컬] 헐 소스파일이름이 255가 넘어??"<<endl;
22. exit(-1);
23. }
25. strncpy(meminfo.filename, filename, strlen( filename ));
26. //실무에서는 더 좋은 함수 쓰세요..
28. CurMemAllocationInfoMap.insert(make\_pair(pfs, meminfo));
29. ++CurMemAllocationCount; // 메모리 할당 수 증가
31. **return** pfs;          // 할당된 주소 반환
32. }
34. **void** operator **delete**(**void** \*p)
35. {
36. // 맵에서 메모리 할당 정보 제거
37. MEN\_INFO\_MAP::iterator it = CurMemAllocationInfoMap.find(p);
38. **if**( CurMemAllocationInfoMap.end() == it )
39. {
40. cerr<< "메모리가 할당된 정보가 없는 주소에 대한 삭제 요청 주소는 : "<< p << endl;
41. **return**;
42. }
44. CurMemAllocationInfoMap.erase(it); // 맵에서 메모리 할당 정보 삭제
45. --CurMemAllocationCount; // 메모리 할당 수 감소
47. free(p); // 메모리 해제
48. }
50. // 현재 메모리 할당 정보가 보고 싶은 시점에 콘솔에 찍어주던지.. 파일로 출력하던지.. EMS나 NMS로 전송하던지.. 입맛에 맞게 추가하시면 됩니당
51. **void** printCurMem()
52. {
53. MEN\_INFO\_MAP::iterator it = CurMemAllocationInfoMap.begin();
54. **while**( CurMemAllocationInfoMap.end() != it )
55. {
56. //TODO
57. //여기에 입맞에 맞게 몇월 몇일 몇시에 [filename]소스파일의 [line]라인에서 [addr] 주소가 할당된 동적 메모리가 현재 남아 있다를 출력해주면 됨
59. ++it;
60. }
62. }

new호출 할 때마다 파일명과 라인 넣기 귀찮으시다구요?

걱정 마세요. 우리에겐 매크로 함수가 있잖아요? :)

요렇게 하나 넣어줍니다.

·미리보기 | 소스복사·

1. **#define new new(\_\_FILE\_\_, \_\_LINE\_\_)**

그리고 일반적인 new 사용하듯이 사용하시면 됩니다.

아참 배열 할당도 정의해 주어야지요 :) 아래와 같은 예로 정의 하시고 입맛에 맞게 수정하여 사용하시면 되겠습니다.

·미리보기 | 소스복사·

1. **void** \*operator **new**[](**size\_t** size, **const** **char** \*filename, **int** line)
2. {
3. **return** operator **new**(size, filename, line);
4. }
6. **void** operator **delete**[](**void** \*p)
7. {
8. operator **delete**(p);
9. }

오늘 다룬 new 및 delete를 오버로딩 하면 다음과 같은 장점이 있어요.

1. 예외에 강해짐.

   (프로그래머가 라이브러리 외부에서 잘못된 포인터를 사용하게 되어도 죽지 않으며, 무려 어느파일 어느라인에서 잘못되었는지 추정가능한 의미있는 데이터를 출력해줍니다!)

2. 메모리가 할당 된 시간을 알 수 있기 때문에 DB내의 데이터와 함께 사용하여, 대부분의 메모리 관련 원인을 찾아 낼 수 있음

3. 동적 메모리 할당 현황에 대한 통계를 프로그래머가 보고 싶은 시점에 언제든지 볼 수 있음

과도한 연산자 오버로딩을 싫어하지는 분도 많으니..

자기 스타일에 맞게 필요할 때! 적재적소에! 사용하시면 되겠습니다 :)

**Why overload new and delete?**

1. To take charge or control over how to allocate memory
2. To aid in debugging; keep track of memory allocation and deallocation in the program
3. To do some other operation apart from allocating memory at the time of memory allocation/decallocation

# [항목9] 리소스 누수를 피하는 방법의 정공은 소멸자다.

예외 처리에 대한 내용은 ec++[항목29] 참고

## 바른 프로그래밍을 위한 고수의 조언

1. 에러 처리에는 예외를 사용하자.   
2. 지역적 흐름 제어 구조로도 충분한 곳에는 예외를 사용하지 말 것.   
3. 자원 관리에는 "자원획득 즉 초기화"(Resource Acquisition is Initialization, RAII) 기법을 사용하자.   
4. 모든 프로그램에 예외 안전성을 부여할 필요는 없다.   
5. 불변속성을 유지하는 데도 "자원획득 즉 초기화" 기법 및 예외 처리자를 사용 하자.   
6. try 블록의 사용은 최대한 자제하자, 직접 에러 처리자 코드를 두기 보다는 "자원획득 즉 초기화" 방법을 적극적으로 활용하자.   
7. 모든 함수에서 모든 가능한 에러를 도맡아 처리할 필요는 없다.   
8. 생성자가 수행되다 오동작이 나면, 예외를 던져서 이것을 알릴 것.   
9. 대입 연산 중에 예외를 던지기 전에, 모든 연산자가 유효한 상태에 있도록 하자.   
10. 소멸자에서는 예외 발생을 피하자.   
11. 모든 예외를 받아 보고하는 코드는 main()에 두도록.   
12. 에러 처리와 상관 없는 코드와 에러 처리 코드는 격리 시킬 것.   
13. 생성자에서 예외를 던지려 할 때 미리 생성자에서 자원이 획득된 상태라면, 해당 자원을 모두 해제하자.   
14. 자원 관리는 단계적으로 진행되도록 하자.   
15. 주로 사용하게 될 인터페이스에 대해서는 예외 지정 기능을 사용하자.   
16. new에 의해 할당되었는데 예외 발생으로 인해 해제되지 않은 메모리가 메모리 누수를 일으킨다는 사실을 잊지말자.   
17. 함수에서 발생할 수 있는 예외는 언젠가 발생할 것이란 가정을 항상 빠뜨리 지 말도록.   
18. 무릇 예외라면 exception 클래스에서 파생되어야 한다는 편견을 버리자.   
19. 라이브러리는 자신의 판단만으로 바로 프로그램을 끝내 버리게 만들면 안 된다. 예외를 발생시켜 라이브러리 호출자에게 판단을 넘기자.   
20. 라이브러리는 최종 사용자를 염두에 둔 진단 출력 메시지를 만들어서도 안 된다. 예외를 발생시켜 호출자에게 판단을 넘기자.   
21. 예외 처리 전략은 설계 단계에서 세우자.

# [항목10] 생성자에서는 리소스 누수가 일어나지 않게 하자.

## 주제: 생성에서의 예외처리

메모리 할당이라는 문제가 항상 곁들여져 있고 생성은 중요한 단계이므로 이를 확실하게 처리하는 방법을 알고 있어야 한다.

## 문제 제기: 객체 생성시, 발생하는 예외

new연산자로 객체를 생성하면 operator new의 메모리 할당과 생성자가 호출된다. 여기서 발생하는 예외를 똑똑히 처리해야 한다.

## 해결

2가지의 과정을 거치므로 각 과정에 대해 논한다.

### 메모리 할당 실패

스마트 포인터를 사용하거나, 그러한 RAII클래스를 만들어서 자원을 관리한다.

<예시 코드>

### 생성자 호출 실패

생성자에서 발생하는 예외이다. 문제는 생성자에서 예외가 발생하면 메모리 할당은 끝난 상태인데 생성은 실패한 셈이므로 소멸자가 호출되지 않는다. 소멸자에서 포함하고 있는 또 다른 객체의 해제를 담당하고 있었다면 메모리는 찔끔찔끔 세어나갈 것이다.

어떻게 처리하면 좋을까?

* 생성자 호출 실패 시, 하염없이 망가져버릴 코드

class Archer

{

public:

Archer(std::string \_name) : name(\_name)

{

wp = new Weapon;

sk = new Skill;

wd = new Wisdom;

}

~Archer()

{

delete wp;

delete sk;

delete wd;

}

private:

std::string name;

Weapon\* wp;

Skill\* sk;

Wisdom\* wd;

};

int main()

{

auto me = new Archer("ytk");

return 0;

}

노란색 부분에서 생성에 실패하면 Archer은 소멸자가 호출되지 않으므로 내부 객체들은 메모리가 누출된다.

* try-catch로 예외를 잡은 생성자

class Archer

{

public:

Archer(std::string \_name) : name(\_name)

{

try

{

wp = new Weapon;

sk = new Skill;

wd = new Wisdom;

}

catch (std::exception& e)

{

destroy();

throw; // 예외를 전파합니다.

}

}

~Archer()

{

destroy();

}

// 코드 중복 방지

void destroy()

{

delete wp;

delete sk;

delete wd;

}

private:

std::string name;

Weapon\* wp;

Skill\* sk;

Wisdom\* wd;

};

int main()

{

try

{

auto me = new Archer("ytk");

}

catch (std::exception& e)

{

std::cerr << e.what();

}

return 0;

}

이렇게 하면 생성자에서 예외가 발생하는 경우, 소멸자 대신 사후처리를 하고 예외를 던진다.

하지만 문제가 하나 더 있다. **멤버가 상수일 경우,** 반드시 초기화 리스트에서 초기화해야 하는데 이는 표현식(expression)이고 try-catch는 문장(statement)이므로 위의 방식대로 처리할 수 없다.

* 스마트 포인터로 각 멤버들이 스스로 자원을 관리한다.

class Archer

{

public:

Archer(std::string \_name) : name(\_name),

wp(new Weapon), sk(new Skill), wd(new Wisdom)

{

}

~Archer()

{

// 할 게 없음

}

private:

std::string name;

const std::unique\_ptr<Weapon> wp;

const std::unique\_ptr<Skill> sk;

const std::unique\_ptr<Wisdom> wd;

};

위에서의 문제를 해결했을 뿐만 아니라, 모든 것이 깔끔하다.

이렇게 해서 생성에 대한 예외 처리는 고민이 끝났다.

# [항목11] 소멸자에서는 예외가 탈출하지 못하도록 하자.

## 주제: 소멸자에서의 예외

왜 소멸자에서의 예외가 중요한가?

## 문제 제기: 소멸자에서 발생하는 예외

예외는 프로그램 작동 중에 문제가 생긴 것이다. 이를 처리하기 위해서 가장 간단한 방법은 예외를 잡는(catch)것이다.

### 스택 풀기

#include <iostream>

using namespace std;

class DividedBy0{

public:

DividedBy0(int p, int q):dividend(p),divisor(q){}

int getDividend(){ return dividend; }

int getDivisor(){ return divisor; }

private:

int dividend, divisor;

};

double getAverageScore(int students, int scores){

if(students == 0) throw DividedBy0(scores, students);

return scores/(double)students;

}

void printAverageScore(int students, int scores){

cout << "평균은 " << getAverageScore(students, scores) << "점입니다." << endl;

}

int main(){

int students, scores;

cout << "학생 수를 입력하세요: ";

cin >> students;

cout << "점수의 합을 입력하세요: ";

cin >> scores;

try{

printAverageScore(students, scores);

}

catch (DividedBy0 e){

cout << "EXCEPTION: " << e.getDividend() << "(을)를 ";

cout << e.getDivisor() << "(으)로 나눴습니다." << endl;

}

return 0;

}

**printAverageScore()** 함수 안에서 나누는 부분을 **getAverageScore()** 함수로 떼어냈고

여기서 throw가 발생하도록 쪼갰습니다.

이때는 예외가 발생하면 **getAverageScore()** 함수가 스택에서 pop되고,

그걸 불렀던 **printAverageScore()** 함수가 pop되고, 또 그걸 불렀던 **main()** 함수까지 가서야

try 문을 찾고 그에 붙어 있는 catch 문으로 들어갑니다!

학생 수를 입력하세요: 0[엔터]

점수의 합을 입력하세요: 417[엔터]

EXCEPTION: 417(을)를 0(으)로 나눴습니다.

[커서]

예외 처리가 아주 잘 됩니다.

이렇게 try 문을 찾을 때까지 스택을 pop하는 것을 **스택 풀기**(stack unwinding)이라 합니다.

이 과정에서 함수 호출 관계가 다 사라져 버리지만, 애초에 예외가 발생했다는 점에서

그 상황이 정상적인 상황은 아닌지라...

그런데 한 번 catch를 한 다음에, 다시 throw를 하는 것도 가능합니다.

#include <iostream>

using namespace std;

class DividedBy0{

public:

DividedBy0(int p, int q):dividend(p),divisor(q){}

int getDividend(){ return dividend; }

int getDivisor(){ return divisor; }

private:

int dividend, divisor;

};

double getAverageScore(int students, int scores){

if(students == 0) throw DividedBy0(scores, students);

return scores/(double)students;

}

void printAverageScore(int students, int scores){

try{

cout << "평균은 " << getAverageScore(students, scores) << "점입니다." << endl;

}

catch(DividedBy0 e){

cout << "EXCEPTION: " << e.getDividend() << "(을)를 ";

cout << e.getDivisor() << "(으)로 나눴습니다." << endl;

throw;

}

}

int main(){

int students, scores;

cout << "학생 수를 입력하세요: ";

cin >> students;

cout << "점수의 합을 입력하세요: ";

cin >> scores;

try{

printAverageScore(students, scores);

}

catch (DividedBy0 e){

cout << "평균 점수를 구하지 못했습니다." << endl;

}

return 0;

}

catch 문 안에서 단순히 그냥 "throw;"라고 써서, 자신이 받았던 예외를 다시 throw하는 경우입니다.

이러면 그 예외를 다시 바깥에서 받아서 또 catch합니다!

여기서는 **getAverageScore()** 함수에서 throw한 예외를 **printAverageScore()** 함수의

catch 문에서 받고, 또 throw해서 **main()** 함수의 catch 문에서 받는 겁니다!

그냥 "throw;"만 쓸 경우 예외 객체는 그대로이므로 자료형도 동일합니다.

학생 수를 입력하세요: 0[엔터]

점수의 합을 입력하세요: 417[엔터]

EXCEPTION: 417(을)를 0(으)로 나눴습니다.

평균 점수를 구하지 못했습니다.

[커서]

두 catch 문의 결과가 차례대로 모두 나옵니다! 순서도 맞습니다.

**[출처]** [[C++ 강좌] 110 - 예외 처리 (5) - 스택 풀기, 다시 throw하기](http://blog.naver.com/kks227/220409398373)|**작성자** [라이](http://blog.naver.com/kks227)

어떤 함수에서 예외가 발생하면 그 함수를 담고 있던 스택이 풀리면서 예외를 잡아줄 장치를 찾는다. 그 과정에서 풀릴 함수 안에 있던 객체들은 소멸한다. 그 소멸하는 과정에서 예외가 또 발생하면 C++은 terminate()를 호출하는 것이다. 그러므로 소멸자에서 예외를 던질 생각을 하면 안 되는 것이다.

### 소멸 중지

소멸자 호출 중에 예외가 발생하면 소멸이 끝나지 않고 범위를 벗어나므로 실행이 끝나지 않은 상태로 남게 된다. 이는 딱 봐도 많은 문제를 품을 것 같다.

## 해결

**[가장 좋은 해결책] 예외를 다른 함수가 담당한다.**

사용자에게 에러를 처리할 수 있는 기회를 주는 것이다. 다음과 같다.

|  |
| --- |
| **class** **DBConnection**{  **public**:  **static** DBConnection create();  void close(); *//close를 호출해야 객체를 소멸한다.*  };  **class** **DBConn**{  **public**:  ***//DBConn에서 DBConnection의 객체를 닫아주는 별도의 다른 함수***  void close(){  *//사용자의 몰지각함에 close호출을 까먹는 경우에 대해*  *//소멸시 자동으로 객체를 닫아줍니다.*  db.close();  closed = true;  }  ~DBConn(){  **if** (!closed)  **try**{  db.close();  }  **catch** (exception e){  *//여기서 프로그램을 종료하거나*  *//abort();*  *//예외를 삼킬 수 있습니다.*  }  }  **private**:  bool closed;  DBConnection db;  };  **중요한 점은 예외는 소멸자가 아닌 다른 곳에서 발생해야 한다는 것.** |

기본적으로 소멸자 내부에서 예외를 잡는다. 그것은 예외가 빠져나가지 못하게 한다는 점에서 어느 정도 문제를 해결(스택 풀기)하지만 여전히 완전히 소멸하지 않는다. 그러므로 이를 다른 일반 함수에서 담당하고, 그 결과를 가지고 상대적으로 안전하게 뒤를 처리할 수 있다.

가장 중요한 핵심은 소멸자는 온전히 끝나야 한다는 것이다. 이에 완전하게 예외를 방지해야 하는데 그 방법을 기억해야 한다.

# [항목12] 예외 발생이 매개변수 전달 혹은 가상 함수 호출과 어떻게 다른지를 이해하자.

## 주제: 예외 발생시의 매개변수 전달

예외를 던지면 catch문은 그것을 받게 된다. 이 형태는 다음과 같다.

class Archer : public std::exception

{

public:

std::string what()

{

return "archer exception.";

}

};

void firstTest()

{

try

{

Archer ytk;

throw ytk;

}

catch (Archer a)

{

throw;

}

}

int main()

{

try

{

firstTest();

}

catch (std::exception e)

{

std::cerr << e.what() << '\n';

}

return 0;

}

이는 함수가 매개변수를 받는 형태와 비슷하지만 다른 점이 있다. 그러므로 명확히 구분하여 문제없이 사용해야 할 것이다.

## 문제 제기: 함수 매개변수와 throw 매개변수의 차이

### 차이점1: 프로그램 흐름

서브루틴(함수)는 내부 작업이 모두 끝나면, 이전 루틴에서 시작한 시점으로 돌아가게 되지만 throw는 돌아가지 않는다. 예외를 던진 순간, 그 함수는 끝난 것이다.

### 차이점2: 객체의 복사

함수의 매개변수는 참조 전달 시 객체를 복사하지 않고 주소를 넘기지만, throw는 무조건 복사한 후에, catch문으로 전달하게 되어있다. 예외를 던진 순간, 그 함수는 모두 죽었다고 했었다. 그러므로 그 안에 있던 모든 객체들은 이미 없는 것이므로 당연한 결과이다. 없는 객체의 참조를 어떻게 전달할까?

### 차이점3: 타입 변환

위에서 제시한 코드를 다시 보자.

class Archer : public std::exception;

void firstTest()

{

try

{

Archer ytk;

throw ytk;

}

catch (Archer a)

{

throw;

}

}

int main()

{

try

{

firstTest();

}

catch (std::exception e)

{

std::cerr << e.what() << '\n';

}

return 0;

}

Archer타입을 매개변수로 예외를 던졌다. 이는 std::exception을 상속받으므로 위에서 2개의 catch문은 모두 이를 받을 수 있다. 그럼 catch가 받은 이 매개변수는 Archer일까? std::exception일까?

객체의 다형성이 가능한 이유는 실제의 타입을 명시한 타입과 구분할 수 있기 때문이다. 위에서도 실제로는 Archer타입을 가져야 맞을 것이지만 catch문에서는 이를 명시한 그대로 판단한다. 그러므로 결과를 호출하면 std::exception의 함수를 호출하게 된다.

void firstTest()

{

Archer ytk;

try

{

std::exception& e = ytk;

throw e;

}

catch (std::exception e)

{

std::cout << "first exception.\n";

throw;

}

}

이러한 경우에도 e는 std::exception형처럼 보이는 Archer지만 catch문은 이를 그저 std::exception로 인식한다는 것이다. 다형성의 원리가 적용되지 않는 것이다.

이것이 암시하는 바는 throw~catch문에서는 암시적 변환이 일어나지 않는다는 것이다. 예를 들어 int->double로의 암시적 변환도 일어나지 않는다. 만약 int를 던졌는데 int를 받는 catch가 없다면 double을 받는 catch문이 있어도 무시하게 된다.

그러므로 다음과 같은 코드에서,

void firstTest()

{

Archer ytk;

try

{

int a = 5;

throw a;

throw ytk;

}

catch (std::exception e)

{

std::cout << "first exception.\n";

throw;

}

catch (double d)

{

std::cout << "first exception.\n";

throw;

}

}

int main()

{

try

{

firstTest();

}

catch (std::exception e)

{

std::cout << "second exception.\n";

std::cerr << e.what() << '\n';

}

return 0;

}

위의 코드는 예외를 받지 못한다. double형의 catch는 int를 받을 수 없다. 하지만 std::exception은 archer을 받을 수 있는데, 이는 상속 관계에 있는 예외들끼리는 형변환이 가능하다는 것을 말해준다.

결국, 예외 전달에서의 타입 변환은 일반적인 암시적 변환은 무시되며 다음과 같은 경우에서 일어난다.

1. 상속
2. void포인터

### 차이점4: 다형성

위에서의 내용과 이어진다. 상속으로 타입 변환이 가능하다고 했는데, 일반적인 함수가 자신의 타입에 맞는 가상 함수를 찾는 과정과는 다르다. 가장 먼저 있는 catch문을 찾아 그에 예외처리가 일어난다.

void firstTest()

{

Archer ytk;

try

{

throw ytk;

}

catch (std::exception e)

{

std::cout << "first exception.\n";

throw;

}

catch (Archer a)

{

std::cout << "first archer exception.\n";

throw;

}

catch (double d)

{

std::cout << "first exception.\n";

throw;

}

}

ytk에 가장 적합한 catch (Archer a)는 이 예외를 절대로 받지 않는다. 가장 먼저 등장한 catch (std::exception e)에서 예외를 처리하게 된다.

# [항목13] 발생한 예외는 참조자로 받아내자.

예외는 참조자로 전달해야 한다. 가장 효율적인 방법이다.

## 주제: 예외 전달

### 포인터 전달

전달하기 전의 객체의 포인터를 전달하므로 전의 객체가 반드시 살아있어야 한다. 하지만 예외는 발생 시 발생한 그 함수는 죽어버리므로 그 안에 있던 객체들은 없는 것이므로 이는 힙이나 전역으로 할당해야 하는데 메모리 관리가 어렵다.

### 값 전달

예외 객체는 2번씩 복사되어야 한다. 왜 2번이나 복사해야 할까?

throw 객체, catch(객체)

던질 때 한번, 받을 때 한번 더 복사하여 2번 복사한다.

슬라이스 문제: 어떤 객체 A를 파생하는 B라는 객체를 예외로 발생시킬 때, 이를 catch에서 A로 받아 복사하게 되면 B라는 객체를 넘길지라도 다형성을 갖지 못하고 그저 A가 되는 것이다.

### 참조자 전달

복사는 던질 때 한번뿐이 발생한다. 그리고 한번 복사하게 되므로 메모리에 대한 걱정을 할 필요도 없다. 최적의 방안인 셈이다.

## 해결

예외는 무조건 참조자로 전달하자.

# [항목14] 예외 지정 기능은 냉철하게 사용하자.

C++에는 임의의 함수에 예외를 지정(exception specification)하는 기능이 있다.

void func(int a, int d) throw(char \*)  
  
이 선언에 의해 func 함수는 char \*형의 예외를 던진다는 것을 알 수 있다. 가능한 예외의 종류가 두 가지 이상일 경우 괄호 안에 예외의 타입들을 콤마로 구분해서 나열한다. 다음 예는 문자열 예외와 정수형 예외를 던지는 함수 func의 원형이다.  
  
void func(int a, int d) throw(char \*, int)  
  
예외를 던지지 않는 함수는 throw()만 적고 괄호 안을 비워 둔다. 함수 원형 뒤에 아무것도 적지 않으면 임의의 예외를 던질 수 있다는 뜻이다. 그래서 다음 두 함수의 뜻은 완전히 다르다.  
  
void func(int a, int d) throw()

## 주제: 예외 지정

어떠한 함수를 정의함에 있어 특정 예외를 명시해 둔다면 보기 좋을 것이다. 그 예외에 대해서만 고려하면 되기 때문이다. 하지만 세상은 그리 호락호락하지 않다. 무엇이 문제인가?

## 문제: 허접한 예외 지정

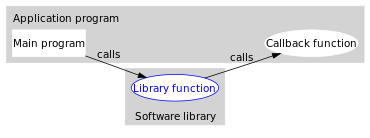
예외를 명시하려다 엉뚱한 예외를 방지하지 못해 흐느적한 코드가 되 버릴 수 있다.

### 템플릿에서의 예외 지정

다양한 타입에 대해 모두 예외를 대비할 수 있을까? 불가능하다. 함수 템플릿에서는 어떠한 타입이 들어올지 모르므로 어떤 기능이 있는지도 모르는데 예외를 고려할 수 있을 리가 없다. 모든 타입에 대해 적용되는 연산(참조자 등)만 들어있을 수도 있다고? 병신. 그 연산이 임의의 타입에서 오버로딩 되어있을 수도 있다.

### 콜백 함수

콜백 - 위키백과, 우리 모두의 백과사전.

[](https://commons.wikimedia.org/wiki/File:Callback-notitle.svg?uselang=ko)

[프로그래밍](https://ko.wikipedia.org/wiki/%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D)에서 **콜백**(callback)은 다른 코드의 인수로서 넘겨주는 [실행 가능한 코드](https://ko.wikipedia.org/wiki/%EC%8B%A4%ED%96%89_%ED%8C%8C%EC%9D%BC)를 말한다. 콜백을 넘겨받는 코드는 이 콜백을 필요에 따라 즉시 실행할 수도 있고, 아니면 나중에 실행할 수도 있다.

일반적으로 콜백수신 코드로 콜백 코드(함수)를 전달할 때는 콜백 함수의 [포인터](https://ko.wikipedia.org/wiki/%ED%8F%AC%EC%9D%B8%ED%84%B0) (핸들), [서브루틴](https://ko.wikipedia.org/wiki/%EC%84%9C%EB%B8%8C%EB%A3%A8%ED%8B%B4) 또는 [람다함수](https://ko.wikipedia.org/wiki/%EB%9E%8C%EB%8B%A4%EB%8C%80%EC%88%98)의 형태로 넘겨준다. 콜백수신 코드는 실행하는 동안에 넘겨받은 콜백 코드를 필요에 따라 호출하고 다른 작업을 실행하는 경우도 있다. 다른 방식으로는 콜백수신 코드는 넘겨받은 콜백 함수를 '핸들러'로서 등록하고, 콜백수신 함수의 동작 중 어떠한 반응의 일부로서 나중에 호출할 때 사용할 수도 있다 (비동기 콜백). 콜백은 [폴리모피즘](https://ko.wikipedia.org/wiki/%ED%8F%B4%EB%A6%AC%EB%AA%A8%ED%94%BC%EC%A6%98)과 [제네릭프로그래밍](https://ko.wikipedia.org/wiki/%EC%A0%9C%EB%84%A4%EB%A6%AD%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D)의 단순화된 대체 수법이며, 콜백 수신 함수의 정확한 동작은 콜백 함수에 의해 바뀐다. 콜백은 [코드 재사용](https://ko.wikipedia.org/wiki/%EC%BD%94%EB%93%9C_%EC%9E%AC%EC%82%AC%EC%9A%A9)을 할 때 유용하다.

콜백 함수로 등록한 함수 포인터에 예외 지정을 하면 그 콜백할 함수가 예외를 일으킬 수 있기 때문에 주의해야 한다(컴파일 에러).

### unexpected

위에서 설명한 예외 지정에 지정되지 않은 예외가 발생하면 어떨까? set\_unexpected로 등록된 함수를 호출하거나 등록되어있지 않다면 unexpected함수를 호출한다.

[std::unexpected\_handler](http://en.cppreference.com/w/cpp/error/unexpected_handler) set\_unexpected( [std::unexpected\_handler](http://en.cppreference.com/w/cpp/error/unexpected_handler) f );

Makes f the new global std::unexpected\_handler and returns the previously installed std::unexpected\_handler.

참고로 std::unexpected\_handler는 typedef void (\*unexpected\_handler)();이다. 함수 포인터다.

## 해결

set\_unexpected의 사용

예상치 못한 모든 예외를 받아 특정 예외로 반환한다.

Set -> my\_unexpectedFunc -> throw myException -> 예측하고 받아내기

# [항목15] 예외 처리에 드는 비용에 대해 정확히 파악하자.

## 주제: 예외 처리 비용

try~catch는 비용을 많이 요구할까? 예외가 발생했을 시, 예외를 throw로 다시 던졌을 때의 성능은 어떠한가?

## 문제: 예외 처리 비용 이해

<http://yesarang.tistory.com/371>

이상을 종합해 보면 다음과 같이 결론 내릴 수 있을 것 같습니다.

**1. 단순히 예외 처리를 위해 try/catch 구조를 사용한다고 하여 overhead가 발생하지는 않는다.**

**2. 예외 발생/처리 overhead는 상당히 크다.**

이 결론은 우리에게 시사하는 바가 상당히 크다고 생각합니다.

1) 예외가 성능을 많이 저하시킨다는 막연한 생각에 아예 예외를 쓰지 않는 것도 피해야 할 것이고, 2) 예외를 예외 답지 않게 사용하는 것-즉, 에러 상황이 아닌 상당히 자주 발생할 수 있는 상황을 예외로 처리하는 경우-도 피해야 할 것이고, 3) 실시간성이 중요한 프로그램에서 호출 경로상 예외가 발생했을 경우 실시간성 검증 없이 예외를 사용하는 경우도 피해야 할 것입니다.

# 효율(Efficiency)

# [항목16] 뼛속까지 잊지 말자, 80-20법칙

프로그램 전체 수행 성능은 수행 코드의 20%에 의해 좌우된다.

## 주제: 최적화

최적화란, 프로그램의 성능을 향상시키는 것.

* 알고리즘 자체에서 고려될 점
* 언어 자체에서 고려될 점

암달의 법칙: p만큼의 부분에서 s만큼의 성능 향상이 있다면 전체 시스템에서는 최대 1/(1-p) + p/s만큼의 성능 향상이 있다.

결론은 부분적으로 아무리 빨라 봐야 오십보백보.

## 문제: 어디서 성능을 개선할까?

병목현상이 일어난다. 즉, 전체 프로그램은 하나의 파이프라인을 통하여 흐르게 되는데 특정 부분만 빠르다고 해서 전체 프로그램이 빨라지는 것은 절대 아니다. 뒤에서 막혀버리면 앞에서 아무리 빠른들 무슨 소용인가?

### 고려할 점

* 하드웨어
* OS
* 컴파일러
* CPU동작

너무 많다. 하나하나가 내용적 깊이가 장난이 아니다.

## 해결: 프로파일링

프로파일러를 이용하여 병목현상의 원인을 파악한다.

# [항목17] 효율 향상에 있어 지연 평가는 충분히 고려해 볼 만하다.

## 주제: 지연 평가

효율성 향상을 위한 또 하나의 방법이다.

지연 평가:  계산의 결과값이 필요할 때까지 계산을 늦추는 기법

이를 어떻게 사용하는지를 알아야 한다. 다음과 같다.

## 문제&해결: 지연 평가 사용하기

### 참조 카운팅

어떤 공간에 데이터를 복사할 때, 그 내용이 필요하기 전까지는 내용 자체를 복사할 필요는 없다. 참조만 하면 된다는 말이다. 즉, 참조 받는 대상(데이터)이 참조 받는 수를 관리만 한다면 참조 수에 따라 메모리 관리도 가능하며 효율적인 성능도 기대할 수 있다.

[항목29] 참조

### 데이터 읽기와 쓰기 구분하기

읽기는 데이터를 참조만 하면 되며, 쓰기는 그렇지 않다. 그러므로 지연 평가를 사용하면 읽기만 하는 자료구조에서 효율적인 성능을 기대할 수 있다. 이는 함수 자체에서 확인할 수는 없고, 프록시 클래스를 이용하여 이를 구현할 수 있다.

[항목5]의 프록시 참조, [항목30] 참조

### 지연 방식의 데이터 가져오기

어떤 객체에 들어있는 데이터를 읽기 위해서 요구기반 객체 초기화 방법을 사용한다. 이것은 데이터를 포인터로 가리키되, 아직 읽지 않았다면 널 값을 가리키므로 이를 검사하여 사용하면 읽지도 않은 데이터를 읽을 필요가 없다.

스마트포인터를 이용하면 더 안전한 구현이 가능하다.

### 지연 방식의 표현식 평가

지연 평가의 전략: 너무 많은 작업이 소요되면 하지 않는다.

행렬 연산에서 즉시 연산을 하지 않고, 전체 코드에서 필요한 부분만 연산한다. 즉, 더하는 연산이 있다면 이는 연산을 직접 하지 않고 더한다는 표현만 자료구조로 저장하면 될 것이다.

## [참고] vector 예시

1 #ifndef VEC\_H

2 #define VEC\_H

3

4 #include <algorithm>

5 #include <cstddef>

6 #include <memory>

7

8 #ifdef \_MSC\_VER

9 #include "../minmax.h"

10 #else

11 using std::max;

12 #endif

13

14 template <class T> class Vec {

15 public:

16 typedef T\* iterator;

17 typedef const T\* const\_iterator;

18 typedef size\_t size\_type;

19 typedef T value\_type;

20 typedef T& reference;

21 typedef const T& const\_reference;

22

23 Vec() { create(); }

24 explicit Vec(size\_type n, const T& t = T()) { create(n, t); }

25

26 Vec(const Vec& v) { create(v.begin(), v.end()); }

27 Vec& operator=(const Vec&);

28 ~Vec() { uncreate(); }

29

30 T& operator[](size\_type i) { return data[i]; }

31 const T& operator[](size\_type i) const { return data[i]; }

32

33 void push\_back(const T& t) {

34 if (avail == limit)

35 grow();

36 unchecked\_append(t);

37 }

38

39 size\_type size() const { return avail - data; } // changed

40

41 iterator begin() { return data; }

42 const\_iterator begin() const { return data; }

43

44 iterator end() { return avail; } // changed

45 const\_iterator end() const { return avail; } // changed

46 void clear() { uncreate(); }

47 bool empty() const { return data == avail; }

48

49 private:

50 iterator data; // first element in the `Vec'

51 iterator avail; // (one past) the last element in the `Vec'

52 iterator limit; // (one past) the allocated memory

53

54 // facilities for memory allocation

55 std::allocator<T> alloc; // object to handle memory allocation

56

57 // allocate and initialize the underlying array

58 void create();

59 void create(size\_type, const T&);

60 void create(const\_iterator, const\_iterator);

61

62 // destroy the elements in the array and free the memory

63 void uncreate();

64

65 // support functions for `push\_back'

66 void grow();

67 void unchecked\_append(const T&);

68 };

69

70 template <class T> void Vec<T>::create()

71 {

72 data = avail = limit = 0;

73 }

74

75 template <class T> void Vec<T>::create(size\_type n, const T& val)

76 {

77 #ifdef \_MSC\_VER

78 data = alloc.allocate(n, 0);

79 #else

80 data = alloc.allocate(n);

81 #endif

82 limit = avail = data + n;

83 std::uninitialized\_fill(data, limit, val);

84 }

85

86 template <class T>

87 void Vec<T>::create(const\_iterator i, const\_iterator j)

88 {

89 #ifdef \_MSC\_VER

90 data = alloc.allocate(j - i, 0);

91 #else

92 data = alloc.allocate(j - i);

93 #endif

94 limit = avail = std::uninitialized\_copy(i, j, data);

95 }

96

97 template <class T> void Vec<T>::uncreate()

98 {

99 if (data) {

100 // destroy (in reverse order) the elements that were constructed

101 iterator it = avail;

102 while (it != data)

103 alloc.destroy(--it);

104

105 // return all the space that was allocated

106 alloc.deallocate(data, limit - data);

107 }

108 // reset pointers to indicate that the `Vec' is empty again

109 data = limit = avail = 0;

110

111 }

112

113 template <class T> void Vec<T>::grow()

114 {

115 // when growing, allocate twice as much space as currently in use

116 size\_type new\_size = max(2 \* (limit - data), ptrdiff\_t(1));

117

118 // allocate new space and copy existing elements to the new space

119 #ifdef \_MSC\_VER

120 iterator new\_data = alloc.allocate(new\_size, 0);

121 #else

122 iterator new\_data = alloc.allocate(new\_size);

123 #endif

124 iterator new\_avail = std::uninitialized\_copy(data, avail, new\_data);

125

126 // return the old space

127 uncreate();

128

129 // reset pointers to point to the newly allocated space

130 data = new\_data;

131 avail = new\_avail;

132 limit = data + new\_size;

133 }

134

135 // assumes `avail' points at allocated, but uninitialized space

136 template <class T> void Vec<T>::unchecked\_append(const T& val)

137 {

138 alloc.construct(avail++, val);

139 }

140

141 template <class T>

142 Vec<T>& Vec<T>::operator=(const Vec& rhs)

143 {

144 // check for self-assignment

145 if (&rhs != this) {

146

147 // free the array in the left-hand side

148 uncreate();

149

150 // copy elements from the right-hand to the left-hand side

151 create(rhs.begin(), rhs.end());

152 }

153 return \*this;

154 }

155

156 #endif

# [항목18] 예상되는 계산 결과를 미리 준비하면 처리비용을 깎을 수 있다.

이를 과도선행평가(over-eager evaluation)라고 부른다.

## 주제: 과도선행평가

또 하나의 최적화 방법이다.

## 문제&해결: 미리 준비하자.

### 캐싱

한번 사용한 자료는 특정 자료구조(map 등…)에 담아서 재활용한다.

### 미리 가져오기

하나를 가져올 때, 하나만 가져오지 않고 그 주변자료까지 한번에 가져온다. 다른 지역은 여러 번 방문하는 것 보다 한번 방문에 여러 개를 가져오는 것이 것이 더 효과적이기 때문이다.

동적 배열도 그러한 예 중 하나이다. 크기를 할당할 때, 본래 크기의 2배씩 할당한다. 이렇게 함으로서 비교적 작은 크기의 배열에서는 적은 할당을, 많이 늘어나야 하는 배열에서도 충분히 빠르게 작동한다. O(k)에 말이다.

캐시 페이징: 페이징이란 [프로그램](http://www.terms.co.kr/program.htm) 중 자주 사용되지 않는 부분의 작업 메모리를 주기억장치인 [메모리](http://www.terms.co.kr/memory.htm)로부터 보조기억장치인 [하드디스크](http://www.terms.co.kr/harddisk.htm)로 옮기는 방식을 통해, 활용 가능한 메모리 공간을 증가시키기 위한 기법 중 하나이다. 이때, 한번에 옮겨지는 메모리 용량 단위를 [페이지](http://www.terms.co.kr/page.htm)라 부른다.

### **페이징**

가상기억장치를 같은 크기의 블록, 즉 페이지(page)로 나누어 사용하는 기법입니다.

* 프레임과 페이지

프레임은 물리 메모리를 일정한 크기로 나눈 블록이고, 페이지는 가상 메모리를 일정한 크기로 나눈 블록입니다. 페이지가 하나의 프레임을 할당 받아 물리 메모리에 위치함으로, 페이지는 알맹이, 프레임은 알맹이가 들어갈 틀이라고 생각하시면 쉽습니다. 프레임을 할당 받지 못한 페이지는 외부 저장장치, (하드디스크 등등)에 저장됩니다. 프레임과 페이지는 같은 크기로 관리됩니다.

* 페이지 테이블(page table)

페이지 테이블은 프로세스의 페이지 정보를 저장하고 있는 테이블입니다. 하나의 프로세스는 하나의 페이지 테이블을 가지고, 테이블은 페이지 번호를 뜻하는 색인과, 해당 페이지에 할당된 물리 메모리(프레임)의 시작주소를 의미하는 내용인 내용으로 구성됩니다.

* 페이지 테이블 엔트리

페이지 테이블 엔트리는(PTE : PageTable Entry)는 페이지 테이블의 레코드, 즉 각 항목으로 페이지 기본주소(Page base address)와 플래그 비트의 내용이 기록됩니다. 플래그 비트에는 접근 비트(페이지에 대한 접근이 있었는가?), 변경 비트(Dirty bit – 페이지 내용의 변경이 있었는가?), 현재비트(Present bit – 현재 페이지에 할당된 프레임이 있는가?), 읽기/쓰기 비트(Read/Write bit – 읽기/쓰기에 대한 권한을 표시)와 같은 내용이 기록됩니다.

* 페이지의 크기

페이지의 크기는 X86과 amd64에서는 4kb, ia64에서는 8kb를 가집니다.

* 동적 주소 변환

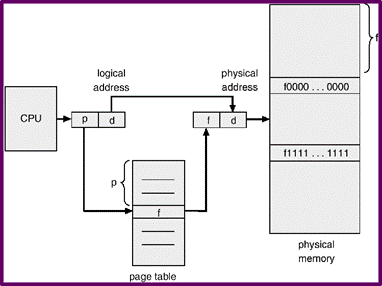
페이징 기법에서 동적 주소 변화는 다음과 같은 과정을 거칩니다.

페이징 기법이 적용된 시스템에서 가상주소는 순써쌍 v(p, d)로 나타나는데, p는 가상기억장치 내에서 참조될 항목이 속해있는 페이지 번호이고, d는 페이지 p 내에서 참조될 항목이 위치하고 있는 곳의 변위입니다.

1) 수행중인 프로세스가 가상 주소 V(p,d) 를 참조

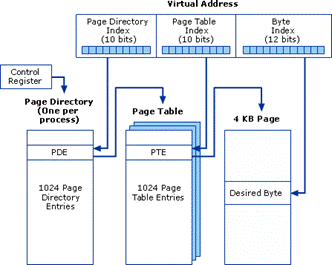
2) 페이징 기법을 통해 페이지 p가 페이지 프레임 P' 에 있음을 알림

3) 실 주소 r = p' + d



### **페이지 디렉토리와 페이지 테이블**

페이징 기능을 사용하지 않았을 때 선형 주소는 물리주소와 같습니다. 그러나 페이징 기능을 사용하면, 논리 주소와 물리 주소의 사이에 선형 주소가 존재하고 따라서 주소를 변경해 주어야 합니다.



페이징을 사용하기 위해서 커널은 미리 페이지 디렉토리와 페이지 테이블을 만들어 놓아야 합니다.

1) 페이지 디렉토리

페이지 디렉토리는 1024개의 디렉토리 엔트리로 구성된 데이터 입니다. 시스템에 하나만 존재합니다. 각각의 디렉토리 엔트리에는 페이지 테이블의 첫 주소, 페이지 테이블의 포인터를 가지고 있습니다. 따라서 시스템에는 1024개의 페이지 테이블이 존재한다고 볼수 있습니다.

2) 페이지 테이블

각각의 페이지 테이블은 또 1024개의 페이지 테이블 엔트리를 가지고 있고 디렉토리 엔트리와 비슷한 구조를 가지고 있습니다. 페이지 테이블의 엔트리는 4KB의 물리 주소, 즉 페이지의 포인터를 가지고 있습니다.

페이지 디렉토리와 페이지 테이블은 다음과 같은 C언어 구문으로 나타낼 수 있습니다.

DWORD page[1024][1024];

전체 페이지 엔트리 수는 1024\*1024 = 1MB 이고, 각각 4KB의 크기를 가지고 있어, 1mb\*4kb = 4GB가 됩니다. 따라서 페이징을 사용하여 4GB 영역의 주소를 지정할 수 있습니다.

4GB영역을 모두 지정하기 위해서는 1024\*4+1024\*4= 16MB(각 엔트리가 32비트)의 영역이 메모리에 할당되고 유지되어야 합니다. 낭비가 될 수 있으므로, 페이지 테이블의 수를 조절합니다.

### **선형주소를 물리주소로**

세그먼트를 이용해 산출된 선형 주소는 1)최상위 10비트는 페이지 디렉토리에서 몇 번째 엔트리를 사용할지 결정하고, 2)그 다음 10비트는 페이지 테이블에서 몇 번재 엔트리를 사용할지 나타내며, 3)하위 12비트는 4KB 물리 페이지상에서 오프셋을 나타냅니다.

CR3 레지스터에는 페이지 디렉토리의 포인터가 저장되어 있습니다. 페이지 디렉토리의 포인터는 처음 페이지를 설정할 때 CR3 레지스터에 설정해 줍니다.

CPU는 프로그램에서 주소 지정을 하면 자동으로 세그먼테이션과 페이징을 수행하여 물리 주소를 계산 합니다. 위의 그림을 참고해주세요.

(1) 물리주소 계산

1) CR3를 참조하여 페이지 디렉토리 찾기

2) 주어진 선형 주소의 최상위 10비트를 참조하여 페이지 디렉토리의 엔트리 번호를 구해, 이 수에 4를 곱한 후 페이지 디렉토리의 포인터와 더해 페이지 디렉토리 내의 엔트리를 찾기

디렉토리 엔트리의 주소 = 선형주소의 최상위 10비트\*4 + 페이지 디렉토리의 포인터

3) 찾아낸 엔트리를 참조하여 페이지 테이블의 물리 주소 찾기

4) 선형 주소의 중간 10비트를 이용하여 페이지 테이블 내의 엔트리 번호 찾기

5) 페이지 테이블에서 물리 페이지의 주소를 구하기

6) 선형 주소의 하위 12비트로 물리 페이지 내에서의 오프셋을 구하여 물리 주소에 접근

페이지 디렉토리 엔트리

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 12 | 11 |  | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 페이지 테이블의 포인터 | | | | | | | | | | | | | | | | | | | | 커널이 사용 | | | G | P  S | 0 | A | P  C  D | P  W  T | U  /  S | R  /  W | P |

페이지 테이블 엔트리

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 12 | 11 |  | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 4KB  물리 페이지의 포인터 | | | | | | | | | | | | | | | | | | | | 커널이 사용 | | | G | P  A  T | D | A | P  C  D | P  W  T | U  /  S | R  /  W | P |

(1) 포인터

페이지 테이블의 포인터와 4KB 물리 페이지의 포인터는 각각 해당하는 물리주소의 4kb를 나눈 값이 들어갑니다. 페이지는 4kb 단위를 기본으로 동작하기 때문입니다. 따라서 페이지 디렉토리나 페이지 테이블, 4KB 물리 페이지는 0x1000단위, 즉 하위 12비트를 0으로 하는 주소에 위치해야 합니다.

(2) A 비트

커널 프로그램이 모든 엔트리의 A 비트를 조사하여 최근에 접근한 페이지나 접근되지 않은 페이지를 찾아냅니다. 페이지에 접근되면 CPU가 자동적으로 해당 엔트리의 A 비트를 1로 세트합니다(클리어는 커널 프로그램의 몫입니다.)

(3) U/S 비트

0으로 클리어 되어 있으면 페이지 테이블도, 4KB 물리 페이지도 커널만 사용할 수 있다는 뜻입니다.

(4) R/W

0으로 클리어 되어 있으면 페이지 테이블이나 4KB 물리 페이지는 읽기만 가능하다는 의미입니다.

(5)P비트

0으로 클리어 되어 있으면 물리 주소상에 페이지가 위치하고 있지 않다는 의미이고, 1이면 물리 주소에 위치하고 있다는 의미입니다.

### **예시**

(1) 선형 주소 0x10065

0x10065를 32비트 이진수로 나타내면 다음과 같습니다.

00000000000000010000000001100101

파란색 부분은 페이지 디렉토리 엔트리 번호이며,

붉은 색 부분은 페이지 테이블의 물리 페이지 엔트리 번호이고,

초록색 부분은 페이지에서의 오프셋입니다.

각각을 16진수로 나누면 다음과 같습니다.

0x0

0x10

0x65

1) CR3에서 페이지 디렉토리의 시작 주소를 구합니다.

2) 페이지 디렉토리의 주소에 페이지 디렉토리 엔트리 번호(선형주소의 상위 10비트)\*4한 값을 더해줍니다. 0이므로 첫번째 엔트리가 될 것입니다.

3) 페이지 디렉토리에서 0번째 엔트리에 포함된 페이지 테이블의 주소를 참조하여 페이지 테이블을 찾습니다.

4) 페이지 테이블의 첫 주소에 0x10\*4=64를 더하여 페이지 테이블 엔트리를 찾습니다. 0x10은 선형 주소의 가운데 10비트로, 페이지 테이블의 물리 페이지 엔트리 번호입니다.

5) 찾은 페이지 테이블 엔트리에서 4kb 물리 페이지의 첫 주소를 찾습니다.

6) 4KB 물리 페이지의 첫 주소에 0x65를 더하여 실제 물리 주소를 구합니다.

페이지 디렉토리와 페이지 테이블을 10비트로 접근 하는 이유는 1024\*4=4069=4KB이기 때문입니다. 반면 페이지의 오프셋을 12비트를 이용하여 접근 하는 이유는 1byte 단위로 접근해야 하기 때문입니다. 따라서 다른 경우 (페이지 디렉토리 엔트리의 페이지 테이블 포인터와 페이지 테이블 엔트리의 4KB 물리 페이지 포인터)와 다르게 4도 곱하지 않습니다.

### **페이지 폴트**

RAM 상에 프로그램이 데이터를 쓸 장소가 없는 경우, 페이지 폴트가 발생됩니다. 이때 14번 인터럽트 핸들러가 실행되고, 커널은 페이지 교체 알고리즘에 따라 디스크의 특정 공간에 저장하고, IRET 명령으로 핸들러를 마친뒤 페이지 폴트가 발생했든 프로그램에게 돌아갑니다.

마찬가지로 RAM상에 프로그램이 찾는 페이지가 없을 경우 (4KB 물리 페이지 엔트리의 p 비트가 0일 때) 페이지 폴트를 발생시키고, 14번 인터럽트를 실행 시켜 디스크의 스왑 영역에서 해당 페이지를 찾아 RAM 상에 옮긴뒤, 다시 해당 프로그램으로 돌아갑니다.

출처: [http://0x200.tistory.com/entry/8-페이징-2-–-페이징](http://0x200.tistory.com/entry/8-%ED%8E%98%EC%9D%B4%EC%A7%95-2-%E2%80%93-%ED%8E%98%EC%9D%B4%EC%A7%95) [0x200.tistory.com]

가상 메모리란 무엇인가? - 페이징 파일의 정체

<http://cappleblog.co.kr/247>

# [항목19] 임시 객체의 원류를 정확히 파악하자

임시 객체가 어디서 생성되는 지를 알아보자는 의미이다.

## 주제: 임시 객체

C++은 실행 도중에 사용자가 원하지 않은 임시 객체를 만든다. 이는 성능 하락을 불러오기 때문에 정확하게 어디서 발생하는지, 어떻게 대처해야 할지 알아야 한다.

## 문제: 임시 객체가 만들어지는 경우

### 함수 호출을 성사시키기 위한 임시객체 생성

매개변수가 다르면 이를 일치시키기 위해서 임시객체를 생성한다. 참고로 참조로 넘기면 임시 객체는 발생하지 않는다. 왜냐하면 임시객체는 본체가 될 수 없기 때문이다.

### 함수가 객체를 값으로 반환할 때

어떤 객체를 반환하면 이는 복사되어야 하므로 새로운 임시객체를 만들어서 반환하게 된다. 기존의 객체는 함수가 끝나면 사라질 것이기 때문이다.

## 해결: 임시객체 생성 방지

### 함수 호출 시에 임시객체 생성 방지

* 1. [항목5] 참고. 코드를 다시 설계해서 타입을 캐스팅이나 매개변수를 바꿔서 맞춰준다.
  2. 타입변환이 불필요하도록 소프트웨어를 수정한다. [항목21] 참고

### 값으로 반환 시의 임시객체 생성 방지

컴파일러의 반환값 최적화를 기대한다. [항목20] 참고

# [항목20] 반환값 최적화가 가능하게 하자.

[항목19]에서 말한 값으로 반환 시의 임시 객체 생성에 대한 최적화를 가능하게 하는 것이다.

## 주제: 반환값 최적화(Return Value Optimization)

## 문제: 임시객체 생성의 비효율성

어떤 계산 결과를 객체로 반환할 시에는 값으로의 전달이 불가피하다.

## 해결

다음과 같이 생성자 인자를 반환하면 컴파일러가 최적화를 시켜준다. 함수가 끝나고 돌아가는 지점에서 객체를 새로 만들지 않아도 되는 것이다. 이를 이어준다고 생각하면 되겠다.

**1. RVO**

최적화를 이용하는 그 방법은 바로 객체 대신에 객체 생성자 인자를 반환하는 것이다.

1. inline const Rational operator \* (const Rational &lhs, const Rational &rhs)
2. {
3. return Rational(lhs.numerator \* rhs.numerator,
4. lhs.denominator \* rhs.denominator);
5. }

얼핏 보기에 위 코드가 무슨 이득이 있나 싶다.

operator \* 안에서 객체가 1번 만들어지고 다시 이것을 반환하면서 임시 객체까지 만들어지니

오히려 손해일 것 같다는 느낌이 팍팍 드는데 말이다.

하지만, 컴파일러는 반환시 임시 객체를 없애고,

계산 결과값을 반환값을 받는 객체에 대해 할당된 메모리에 직접 넣어 초기화해 준다.

결국 생성자 한번, 소멸자 한번의 호출 비용만 들어가는 것이다.

이것이 바로 RVO(Return Value Optimization)인 것이다.

**2. NRVO (Named RVO)**

vs2005부터 지원되는 NRVO는 말 그대로 이름이 있는 변수에 대해서도 RVO가 적용되는 것이다.

RVO와는 다르게 **NRVO는 최적화 옵션 /O1(크기 최소화)부터 동작**함을 기억하자.

바로 예제부터보자.

1. class RVOSample
2. {
3. public:
4. RVOSample();
5. ~RVOSample();
7. int value;
8. };
10. RVOSample TestNRVO(int num)
11. {
12. RVOSample rvo;
13. rvo.value = num;
14. return rvo;    // 임시 객체가 생성되지 않는다.
15. }

RVO와 다르게 변수를 선언하고, 해당 변수를 사용한 다음 반환해도 임시 객체가 생성되지 않는다.

RVO에 비해 훨씬 쓰게 편하고 보기도 익숙하다.

하지만, 아래 예제와 같은 경우엔 NRVO로 최적화되지 않는다.

1. RVOSample TestNRVO(int num)
2. {
3. RVOSample rvo;
4. rvo.value = num;
6. */- 조건에 의해 반환값이 다른 경우 NRVO로 최적화되지 않는다. \*-*
7. if (5 == num)
8. {
9. rvo.value \*= 2;
10. return rvo;
11. }
13. return rvo;
14. }

이 점만 유의하면, 중간에 어떠한 코드들이 들어가도 임시 객체는 생성되지 않는다.

**3. RVO / NRVO 종합 예제**

1. #include "stdafx.h"
2. #include <iostream>
4. using namespace std;
6. class RVOTest
7. {
8. public:
9. RVOTest() { cout << "Default Ctor" << endl; }
10. ~RVOTest() { cout << "Dtor" << endl; }
12. RVOTest(const RVOTest& rhs) { cout << "Copy Ctor" << endl; }
13. };
15. RVOTest RVOFunc()
16. {
17. return RVOTest();
18. }
20. RVOTest NRVOFunc()
21. {
22. RVOTest r;
23. return r;
24. }
26. int \_tmain(int argc, \_TCHAR\* argv[])
27. {
28. RVOFunc();
29. NRVOFunc();
31. */-*
32. *위 두 함수의 결과는 아래와 같다.*
33. *Default Ctor*
34. *Dtor*
36. *만약, RVO가 적용되지 않는다면 결과는 다음과 같다.*
37. *Default Ctor*
38. *Copy Ctor -> 임시객체 생성*
39. *Dtor      -> 임시객체 생성*
40. *Dtor*
41. *\*-*
43. return 0;
44. }

# [항목21] 오버로딩은 불필요한 암시적 타입변환을 막는 한 방법이다.

## 문제: 매개변수의 암시적 변환

컴파일러가 알아서 처리하는 타입변환은 임시 객체의 비용이 불가피하다.

## 해결: 오버로딩

타입 변환을 기대하지 말고 변환된 타입 또한 받아들이도록 오버로딩을 한다.

### 주의: 오버로딩되는 연산자 함수는 반드시 최소한 한 개의 사용자 정의 타입을 매개변수로 가져야 한다.

# [항목22] 단독 연산자(op) 대신에 =이 붙은 연산자(op=)를 사용하는 것이 좋을 때가 있다.

x = x + y;와 x += y;을 말하는 것이다.

## 연산자 효율적으로 구현하기

### 오버로딩된 +=연산자 함수 이용하기

#include <iostream>

#include <cstring>

class MyString

{

public:

MyString(char\* s) : mS(s), length(strlen(s)) {}

const char\* getData()

{

return mS;

}

int getLength()

{

return length;

}

const MyString operator+=(MyString& rhs)

{

char\* newS = (char\*)malloc(length + rhs.getLength());

for (int i = 0; i < length; i++)

{

newS[i] = mS[i];

}

auto rhsS = rhs.getData();

for (int i = 0; i < rhs.getLength(); i++)

{

newS[length + i] = rhsS[i];

}

length += rhs.getLength();

newS[length] = '\0';

mS = newS;

return MyString(newS);

}

const MyString operator+(MyString& rhs)

{

return MyString(mS) += rhs;

}

friend std::ostream& operator<< (std::ostream &out, MyString& ms) {

out << ms.getData();

return out;

}

private:

char\* mS;

int length;

};

int main()

{

MyString ms1("ytk");

MyString ms2("iszzang");

ms1 += ms2;

std::cout << ms1 << '\n';

auto ms3 = ms1 + ms2;

std::cout << ms3 << '\n';

return 0;

}

### 임시 객체 최소화

주석 처리된 부분보다 아래가 더 효율적이다. 임시 객체가 필요가 없기 때문이다.

//auto ms3 = ms1 + ms2;

auto ms3 = ms1;

ms3 = ms2;

# [항목23] 정 안되면 다른 라이브러리를 사용하자!

## 문제제기: 라이브러리 성능

라이브러리가 모든 부분을 충족시킬 수 없다. 즉, 속도와 크기에 최적화되어 있다면 이식성이 떨어질 수 있고, 기능이 많으면 직관성이 떨어질 수 있다.

### Printf vs. cout

<http://minjang.egloos.com/1934885>

## 해결: 다른 라이브러리 사용

다른 라이브러리가 있는지 찾아보자. 이것으로 병목현상이 해결될 수 있다.

# [항목24] 가상 함수, 다중 상속, 가상 기본 클래스, RTTI에 들어가는 비용을 제대로 파악하자.

## 가상 함수

### 가상 함수 정의, 구현

출처: <http://kkojabee.tistory.com/entry/More-Effective-C-ItemItem-24-Understand-the-costs-of-virtual-functions-multiple-inheritance-virtual-base-classes-and-RTTI> [For Better Life !]

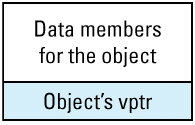
가상 함수가 호출될 때, 실행되는 코드는 그 객체의 동적인 타입에 따라 상승하는 함수를 호출해야 합니다. 컴파일러는 이런 작업을 어떻게 효율적으로 제공할까요 ? 대부분의 컴파일러 구현은 **virtual tables**와 **virtual table pointers**를 사용합니다. 보통 virtual tables는 **vtbls**로 virtual table pointers는 **vptrs**로 불립니다.

class C1 {  
public:  
  C1();  
  
  virtual ~C1();  
  virtual void f1();  
  virtual int f2(char c) const;  
  virtual void f3(const string& s);  
  
  void f4() const;  
  
  ...  
};  


여기에 가상 함수들에 필요한 첫번째 비용이 발생합니다. **가상 함수를 포함한 각 클래스들은 virtual table들을 위한 공간이 별도로 필요합니다.**

…생략…

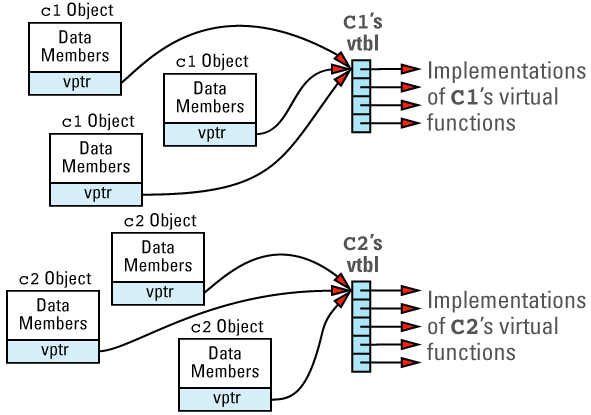
가상 함수들이 선언된 클래스들의 객체는 그 클래스의 virtual table을 지시하는 감춰진 데이터 멤버가를 가지고 다닙니다.  이 감춰진 데이터 멤버 vptr은 그 객체 내부 오직 컴파일러들만 아는 곳에 추가됩니다. 개념적으로 다음과 같이 도시할 수 있습니다.



이 그림에서는 편의상 객체의 vptr을 객체의 끝에 놓았지만, 그 위치는 컴파일러 별로 다름에 유의합니다.  상속이 존재한다면, 객체의 vptr은 데이터 멤버들에 둘러 쌓여 집니다. 다중 상속은 이 그림을 복잡하게 만듭니다. 이 부분은 좀 더 뒤에서 다루기로 합니다. 여기에서는, 가상 함수들의 두 번째 비용에 대해 주목합니다. **가상 함수들을 포함하는 클래스의 각 객체는 추가적인 포인터를 소비합니다.**

### 가상 함수 사용에 드는 비용

여러 개의 C1과 C2 객체를 가진 프로그램이 있다고 가정해 봅니다. 각 객체들 사이에서 vptr들과 vtbl들은 다음과 같이 나타낼 수 있습니다.



지금부터는 이 프로그램 조각을 고려합니다.

void makeACall(C1 \*pC1)  
{  
  pC1->f1();  
}

이것은 포인터 pC1을 통해서 f1 함수를 호출합니다. 이 코드만으로는 f1 함수가 C1::f1인지 C2::f1인지 알 수 없습니다. 컴파일러들은 이 호출을 다음과 같이 진행합니다.  
  
1. 해당 객체의 vptr에서 vtbl 포인터를 얻습니다. 결국, vptr offet 과 포인터 전환 비용만 듭니다.  
2. 호출되는 함수 (여기서는 f1)에 대응되는 포인터를 vtbl에서 찾습니다. 이 또한 컴파일러들이 vtbl 내에 각 가상함수를 유일한 인덱스로 넣어 두었기 때문에 단지 vtbl 배열로의 offset 비용만이 발생합니다.  
3. 2에서 얻은 포인터로 함수를 호출합니다.  
  
이 과정을 감춰진 vptr로 표현하면 다음과 같습니다.

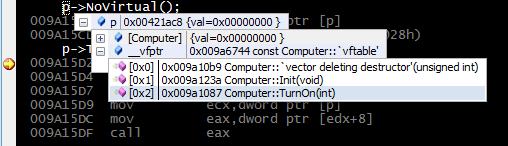
pC1->f1();  
  
(\*pC1->vptr[i])(pC1);

// call the function pointed to by the i-th entry in the vtbl pointed to  
// by pC1->vptr; pC1 is passed to the function as the "this" pointer

이 과정은 거의 비가상 함수 호출만큼이나 효율적입니다. 모든 기기들에서 아주 적은 부가 인스트럭션들로 수행됩니다. 가상함수를 호출하는 비용은 기본적으로 함수 포인터를 통해 함수를 호출하는 비용과 같습니다. 결국, **가상함수들은 대개 성능의 병목은 아닙니다.**  
가상 함수들의 실제 수행 비용은 inlining과 같이 상호작용하는 것과 관련이 있습니다. 모든 실제적인 목적에서, 가상 함수들은 인라인되지 않습니다. 이는 컴파일시 "inline"이 의미하는 것은 "호출 지점을 호출 함수 바디로 대체하라"는 것인데 "virtual"이 의미하는 것은 "어떤 함수가 호출될 지 실행시까지 기다려라"이기 때문입니다. 만약 컴파일러들이 특정 위치에서 어떤 함수들이 호출될 지를 모른다면 우리는 이들이 이 함수 호출을 왜 inline할 수 없는 지도 이해할 수 있을 것입니다. 이것이 가상 함수들의 세번째 비용입니다. **가상 함수들은 인라인하지 않아야 합니다**.

출처: <http://kkojabee.tistory.com/entry/More-Effective-C-ItemItem-24-Understand-the-costs-of-virtual-functions-multiple-inheritance-virtual-base-classes-and-RTTI> [For Better Life !]

첫 번째 NoVirtual 함수 호출은 보다시피 call 할 주소가 바로 계산이 되어있다. 반면, TurnOn이라는 가상 함수는 이 주소를 직접 계산 한다. 00BA15EB에 있는 문장은 C++ 클래스의 [virtual table](http://en.wikipedia.org/wiki/Virtual_table)에 접근하는 부분이다. 이건 당연히 클래스 선언 레이아웃에 따라 달라질 것이다. vtable은 디버거로 쉽게 살펴볼 수 있다. vtable 3번 째 항목에 TurnOn이 있어서 +8 오프셋이 주어졌다.



대충 가상 함수가 어떻게 생겨먹었고 어떻게 불리는지 살펴봤다. 가상 함수를 한 줄로 요약하면

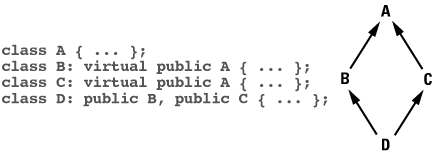
호출할 함수의 대상이 컴파일 시간에 정해지지 않고 실행 시간에 계산을 통해 결정 되는 간접 분기문

으로 요약할 수 있다. 따라서 OOP 언어의 가상 함수 뿐만 아니라 C 언어에서도 함수 포인터를 통한 호출도 이와 동일한 맥락이다. C에서도 함수 포인터를 모아놓은 자료구조로 얼마든지 OOP틱하게 코딩 할 수 있다. 따라서 이 포스팅이 답하고자 하는 질문은 프로그래밍 언어를 막론하고 간접 분기문에 대한 비용은 얼마나 큰가? 로 정리할 수 있다.

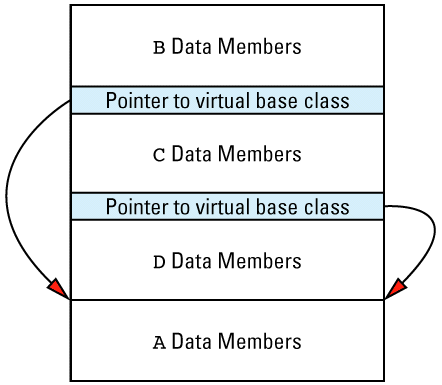
출처: <http://minjang.egloos.com/1973793>

### 다중 상속

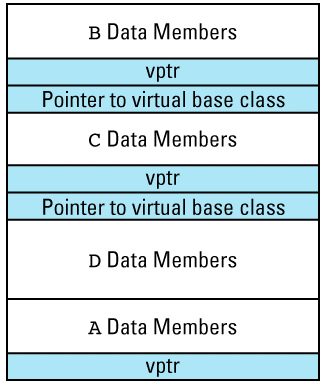
예를 들어, Scott Meyers가 "the dreaded multiple inheritance diamond"라 부르는 문제를 생각해 봅니다.



여기서 A가 가상 기저 클래스입니다.  (특히 예전) 컴파일러들에서 D 타입의 객체는 다음과 같은 구조를 가질 것입니다.



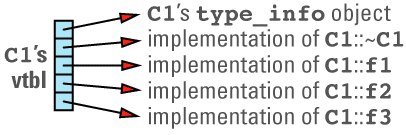
이 그림에서 A의 데이터 멤버들이 가장 끝에 놓인 것이 이상하기는 하지만, 대개 이런 방식으로 구현된다고 합니다.   
  
이 그림과 이전의 vptr 그림과 같이 묶으면 다음의 그림을 얻을 수 있습니다.



이 그림에서 파란색으로 표시된 부분들이 컴파일러에 의해 추가된 것들입니다. 이 그림은 실제 데이터 멤버들의 크기에 따라 비율이 조절되어야만 합니다. 작은 클래스들에서는 이 부하가 비교적 클 것이며, 큰 클래스들에서는 비교적 작을 것입니다.  
  
위 그림에서 이상한 점은 관련된 클래스가 4 개인데 3 개의 vptr 만이 있는 것입니다. 물론 4 개의 vptr로도 구현이 가능하지만 대부분의 컴파일러는 컴파일러에서 발생하는 부하를 줄이기 위해 3 개의 vptr로 구현합니다.  
  
출처: <http://kkojabee.tistory.com/entry/More-Effective-C-ItemItem-24-Understand-the-costs-of-virtual-functions-multiple-inheritance-virtual-base-classes-and-RTTI> [For Better Life !]

### RTTI

각 클래스에 대해 RTTI 정보는 하나만이 필요합니다만, 임의의 객체들에 대해서 이 정보를 획득하기 위한 방법이 필요합니다. 실상, 이는 완전한 사실이 아닙니다. C++ 언어 규격은 그 타입이 최소한 하나 이상의 가상 함수를 가져야만 객체의 동적 타입에 대한 정확한 정보를 보장한다고 언급하고 있습니다. 이는 RTTI 데이터가 virtual function table과 비슷하다는 것처럼 들립니다. 우리는 클래스마다 하나의 정보를 원하며, 가상 함수를 포함한 어떤 객체에서도 적합한 정보를 얻을 수 있는 방법이 필요합니다. 이러한 RTTI와 vtbl의 동질성은 우연이 아닙니다. RTTI는 클래스의 vtbl로 구현되도록 설계되었습니다.  
  
예를 들어, vtbl 배열의 0번째 인덱스를 그 클래스의 type\_info를 가리키도록 할 수 있습니다. 위 C1 클래스의 vtbl은 다음과 같이 표현될 수 있습니다.



이 구현에 의하면, RTTI에 의한 추가 공간 비용은 vtbl에 추가 항목과 type\_info 객체 저장 공간입니다. vtbl을 위한 메모리가 대부분의 어플리케이션들에서 눈에 띄지 않는 것처럼, type\_info 객체들의 크기가 문제가 되지는 않을 것입니다.

출처: <http://kkojabee.tistory.com/entry/More-Effective-C-ItemItem-24-Understand-the-costs-of-virtual-functions-multiple-inheritance-virtual-base-classes-and-RTTI> [For Better Life !]

# [항목25] 생성자 함수와 비멤버 함수를 가상 함수처럼 만드는 방법

가상함수: 어떤 함수가 호출될지를 해당 타입에 따라 동적으로 결정한다.

class Archer

{

public:

Archer() {}

Archer(const UnitData& data)

: level(data.level), power(data.power) {}

~Archer() {}

Archer(const Archer& lhs)

: level(lhs.level), power(lhs.power) {}

Archer& operator=(const Archer& lhs) {}

//가상 복사 생성자

virtual Archer\* clone() const = 0;

virtual void getInfo() const = 0;

virtual std::ostream& print(std::ostream& os) const = 0;

private:

int level{1};

int power{1};

};

class BladeMaster : public Archer

{

public:

BladeMaster(const UnitData& data)

{

}

virtual BladeMaster\* clone() const

{

std::cout << "create blademaster.\n";

return new BladeMaster(\*this);

}

virtual void getInfo() const

{

std::cout << "bowMaster" << std::endl;

}

virtual std::ostream& print(std::ostream& os) const

{

return os << "bladeMaster\n";

}

};

class BowMaster : public Archer

{

public:

BowMaster(const UnitData& data)

{

}

virtual BowMaster\* clone() const

{

std::cout << "create bowmaster.\n";

return new BowMaster(\*this);

}

virtual void getInfo() const

{

std::cout << "bowMaster" << std::endl;

}

virtual std::ostream& print(std::ostream& os) const

{

return os << "bowMaster\n";

}

};

//비멤버 함수가 가상함수처럼 작동한다.

inline

std::ostream& operator<<(std::ostream& os, const Archer& a)

{

a.print(os);

}

class Barrack

{

public:

static Archer\* createArcher(const UnitData& data)

{

if (data.type == UnitData::UnitType::NORMAL)

{

//return new Archer(data);

return nullptr;

}

else if (data.type == UnitData::UnitType::SWORD)

{

return new BladeMaster(data);

}

else if (data.type == UnitData::UnitType::BOW)

{

return new BowMaster(data);

}

}

private:

std::map<std::string, Archer\*> units;

};

int main()

{

UnitData data;

data.type = UnitData::UnitType::SWORD;

data.level = 1;

data.name = "yth";

data.power = 9999;

auto a = Barrack::createArcher(data);

a->getInfo(); // bladeMaster

auto b = reinterpret\_cast<BladeMaster\*>(a)->clone(); // create bladeMaster

auto c = a->clone(); // create bladeMaster

b->getInfo(); // bladeMaster

std::cout << "total info: " << a << b << c;

return 0;

}

## 가상 생성자

다른 타입(포인터 타입)의 객체를 생성하는 함수를 만든다.

static Archer\* createArcher(const UnitData& data)

{

if (data.type == UnitData::UnitType::NORMAL)

{

//return new Archer(data);

return nullptr;

}

else if (data.type == UnitData::UnitType::SWORD)

{

return new BladeMaster(data);

}

else if (data.type == UnitData::UnitType::BOW)

{

return new BowMaster(data);

}

}

**가상 생성자**는 주어진 정보를 이용하여 생성할 객체를 동적으로 결정한다. 기본 포인터를 반환하되, 본래의 객체는 다를 수 있다. 어떤 생성자를 선택할지를 동적으로 결정하므로 가상 생성자이다.

이 생성자(createArcher)를 호출하는 객체는 자기 자신(Archer)이 아니라 일종의 팩토리(Barrack)이다. 이 구조를 잘 기억하자.

## 가상 복사 생성자

class Archer

{

public:

//가상 복사 생성자

virtual Archer\* clone() const = 0;

virtual void getInfo() const = 0;

};

class BladeMaster : public Archer

{

public:

virtual BladeMaster\* clone() const

{

std::cout << "create blademaster.\n";

return new BladeMaster(\*this);

}

virtual void getInfo() const

{

std::cout << "bowMaster" << std::endl;

}

};

**가상 복사 생성자(**clone**)**는 자기 자신을 복사한 객체의 포인터를 반환(new BladeMaster(\*this))한다. 이는 추상 클래스의 함수를 재정의한 가상함수지만 반환 포인터가 다를 수 있다는 C++성질을 이용한다. 어떤 clone을 호출할지는 가상함수이므로 동적으로 결정할 수 있고 반환할 포인터도 동적으로 결정되어 객체에 맞는 포인터를 반환할 수 있다. 뿐만 아니라 객체가 복사 생성자에서 가지는 특별한 기능에 대한 복사도 그대로 가능하게 한다.

## 비멤버 함수를 가상 함수처럼 동작하게 하기

비멤버 함수를 가상 함수처럼 만드는 방법

**필요한 일을 하는 가상 함수를 만들어 놓고, 비멤버 함수에서 해당 가상 함수를 호출하면 된다.**

추가적인 함수 호출의 비용은 인라인으로 해결하면 된다. 위의 예제에도 설명되어 있다.

class Archer

{

public:

virtual std::ostream& print(std::ostream& os) const = 0;

};

class BladeMaster : public Archer

{

public:

virtual std::ostream& print(std::ostream& os) const

{

return os << "bladeMaster\n";

}

};

class BowMaster : public Archer

{

public:

virtual std::ostream& print(std::ostream& os) const

{

return os << "bowMaster\n";

}

};

//비멤버 함수가 가상함수처럼 작동한다.

inline

std::ostream& operator<<(std::ostream& os, const Archer& a)

{

a.print(os);

}

int main()

{

// 아래처럼 표준 출력 객체에 대해서 작동하지는 않습니다.

std::cout << "total info: " << a << b << c;

return 0;

}

위의 예제에서는 virtual std::ostream& print함수를 동적으로 결정한다. 비멤버 함수(std::ostream& operator<<)에서는 이 print함수를 호출할 뿐이다. 그럼 객체가 가상 함수 동작에 맞게 알아서 자기에 맞는 함수를 선택할 것이다.

# [항목26] 클래스 인스턴스의 개수를 의도대로 제한하는 방법

## 1개: 싱글턴

문제점 파악

## 그 이상 혹은 객체 지향: 인스턴스 카운팅

#include <iostream>

#include <memory>

template<typename Object>

class CountingWrapper

{

public:

class TooManyObjects {};

static size\_t objectCount() { return numObjects; }

protected:

CountingWrapper();

CountingWrapper(const CountingWrapper& rhs);

~CountingWrapper();

private:

static const size\_t maxObjects;

static size\_t numObjects;

void init();

};

template<typename Object>

CountingWrapper<Object>::CountingWrapper()

{

init();

}

template<typename Object>

CountingWrapper<Object>::CountingWrapper(const CountingWrapper& rhs)

{

init();

}

template<typename Object>

CountingWrapper<Object>::~CountingWrapper()

{

numObjects--;

}

template<typename Object>

void CountingWrapper<Object>::init()

{

if (numObjects >= maxObjects) throw TooManyObjects();

numObjects++;

}

class Archer : private CountingWrapper<Archer>

{

public:

using CountingWrapper<Archer>::objectCount;

static Archer\* makeArcher()

{

return new Archer();

}

private:

Archer() {}

Archer(const Archer&) {}

};

//template<typename Object>

const size\_t CountingWrapper<Archer>::maxObjects = 10;

size\_t CountingWrapper<Archer>::numObjects = 0;

int main()

{

auto a = Archer::makeArcher();

return 0;

}

# [항목27] 힙(heap)에만 생성되거나 힙에는 만들어지지 않는 특수한 클래스를 만드는 방법

힙?

## 힙에서만 생성되는 객체

스택은 안 된다. 뭘 하려고 그러는 것일까? 스스로 메모리를 해제하는 delete this를 사용하기 위해서다. 힙에서만 할 수 있는 동작을 생성자 혹은 소멸자에서 쓰고 싶은 것이다!

class Archer

{

public:

Archer() {}

void destroy()

{

delete this;

}

protected: // 상속을 위한 선언

~Archer() {}

};

class FuckingCastle

{

public:

FuckingCastle() { unit = new Archer(); }

~FuckingCastle() { unit->destroy(); }

private:

Archer\* unit;

};

int main()

{

try

{

//Archer stack; // error: ~Archer()에 액세스할 수 없음

auto heap = new Archer();

heap->destroy();

}

catch (std::exception& e)

{

e.what();

}

}

이렇게 소멸자를 private으로 선언하면 상속을 방지할 수도 있다. 그리고 합성(인스턴스를 다른 객체에 넣는 일)도 방지한다. 만약 상속은 허용하고 싶다면 위처럼 protected로 선언한다.

## 어떤 객체가 힙에서 생성되었는지 아닌지를 알아내는 방법

결론은 쉽지 않다.

### Static bool을 이용한 flag setting방법

배열, 객체의 주소를 인자로 할당하는 경우에서 안 된다. 정적 변수가 정상적으로 작용할 수 없다.

class Archer

{

public:

Archer() {}

static void\* operator new(std::size\_t size)

{

std::cout << "alloc.\n";

onTheHeap = true;

return ::operator new(size);

}

private:

static bool onTheHeap;

};

bool Archer::onTheHeap = false;

### 스택과 힙이 반대 방향으로 채워진다는 점

이러한 주소 접근 방법은 안 된다. 이식성은 당연히 없을 뿐만 아니라 정적 객체가 힘 아래에서 꿈틀거리기 때문에 불가능하다.

### 힙에 할당했으면 이를 컬렉션에 저장하여 관리하는 방법

이 방법은 힙 주소를 가리키는지 알아내는 방법이 아니라, 해당 포인터가 삭제해도 되는지 알아내는 방법이다.

class Item // 무언가로부터 상속받을 객체

{

public:

virtual ~Item() {}

void\* operator new(std::size\_t size)

{

auto ptr = ::operator new(size);

addresses.push\_front(ptr);

return ptr;

}

bool isOnHeap()

{

const void\* rawAddress = dynamic\_cast<const void\*>(this);

return std::find(addresses.cbegin(), addresses.cend(), rawAddress)

!= addresses.end();

}

private:

using Address = const void\*;

static std::list<Address> addresses;

};

std::list<Item::Address> Item::addresses;

* 주소 컬렉션(std::list<Item::Address> Item::addresses)을 만든다.
* 포인터 주소를 추가(addresses.push\_front(ptr))하는 기본 클래스를 만든다. 그리고 이 기능을 사용할 클래스는 이를 상속한다.
* 힙에 있는지 확인(isOnHeap())하기 위해 주소 컬렉션에 해당 주소가 존재하는지 확인한다.

**여기에는 중요한 문제 해결 원리가 3가지 숨어있다.**

* 어떠한 기능을 가지는 하나의 클래스로 표현한다. 이들을 관리하는 리스트를 그 클래스 안에 넣어서 전역적인 접근을 피하면서 동시에 해당 기능의 명시를 분명하게 한다.
* 다중상속에서 포인터의 주소 값이 다를 수 있다는 문제가 있다. 이는 = dynamic\_cast<const void\*>로 해당 객체의 가장 앞부분의 메모리 주소 값을 얻어와서 시작 위치를 얻어온다는 점을 이용한다.

결론은 상속할 기본 클래스를 만들고 다이나믹 캐스트로 본래의 기본 클래스의 주소를 가져오고 그것을 저장하여 리스트로 관리한다.

## 객체가 힙에 생성되지 않게 하기

Operator new를 어떻게 다룰까라는 생각이 떠오른다. 정답이다. Private에 쑤셔 넣자.

class FuckingCastle

{

public:

FuckingCastle() { unit = new Archer(); }

~FuckingCastle() { unit->destroy(); }

private:

Archer\* unit;

static void\* operator new(std::size\_t size);

static void operator delete(void\* ptr);

};

int main()

{

try

{

FuckingCastle fc;

FuckingCastle\* pFc = new FuckingCastle();

// error: FuckingCastle()에 액세스할 수 없음

}

catch (std::exception& e)

{

e.what();

}

}

# [항목28] 스마트 포인터(Smart Pointer)

원시 포인터를 감싸는 클래스로 여러 가지 일을 포인터 스스로 처리하게 한다.

* 생성, 소멸 작업 조절
* 복사, 대입 동작 조절
* 역참조 동작 조절

## std::unipue\_ptr

00001 // unique\_ptr implementation -\*- C++ -\*-

00002

00003 // Copyright (C) 2008, 2009 Free Software Foundation, Inc.

00004 //

00005 // This file is part of the GNU ISO C++ Library. This library is free

00006 // software; you can redistribute it and/or modify it under the

00007 // terms of the GNU General Public License as published by the

00008 // Free Software Foundation; either version 3, or (at your option)

00009 // any later version.

00010

00011 // This library is distributed in the hope that it will be useful,

00012 // but WITHOUT ANY WARRANTY; without even the implied warranty of

00013 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

00014 // GNU General Public License for more details.

00015

00016 // Under Section 7 of GPL version 3, you are granted additional

00017 // permissions described in the GCC Runtime Library Exception, version

00018 // 3.1, as published by the Free Software Foundation.

00019

00020 // You should have received a copy of the GNU General Public License and

00021 // a copy of the GCC Runtime Library Exception along with this program;

00022 // see the files COPYING3 and COPYING.RUNTIME respectively. If not, see

00023 // <http://www.gnu.org/licenses/>.

00024

00025 /\*\* @file unique\_ptr.h

00026 \* This is an internal header file, included by other library headers.

00027 \* You should not attempt to use it directly.

00028 \*/

00029

00030 #ifndef \_UNIQUE\_PTR\_H

00031 #define \_UNIQUE\_PTR\_H 1

00032

00033 #ifndef \_\_GXX\_EXPERIMENTAL\_CXX0X\_\_

00034 # include <[c++0x\_warning.h](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00611.html)>

00035 #endif

00036

00037 #include <[bits/c++config.h](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00613.html)>

00038 #include <[debug/debug.h](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00697.html)>

00039 #include <[type\_traits](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00928.html)>

00040 #include <[utility](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00946.html)>

00041 #include <[tuple](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00925.html)>

00042

00043 \_GLIBCXX\_BEGIN\_NAMESPACE(std)

00044

00045 /\*\*

00046 \* @addtogroup pointer\_abstractions

00047 \* @{

00048 \*/

00049

00050 /// Primary template, default\_delete.

00051 template<typename \_Tp>

[00052](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00357.html) struct [default\_delete](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00357.html)

00053 {

00054 default\_delete() { }

00055

00056 template<typename \_Up>

00057 default\_delete(const [default\_delete<\_Up>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00357.html)&) { }

00058

00059 void

00060 operator()(\_Tp\* \_\_ptr) const

00061 {

00062 static\_assert(sizeof(\_Tp)>0,

00063 "can't delete pointer to incomplete type");

00064 delete \_\_ptr;

00065 }

00066 };

00067

00068 // \_GLIBCXX\_RESOLVE\_LIB\_DEFECTS

00069 // DR 740 - omit specialization for array objects with a compile time length

00070 /// Specialization, default\_delete.

00071 template<typename \_Tp>

[00072](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00358.html) struct [default\_delete](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00357.html)<\_Tp[]>

00073 {

00074 void

00075 operator()(\_Tp\* \_\_ptr) const

00076 {

00077 static\_assert(sizeof(\_Tp)>0,

00078 "can't delete pointer to incomplete type");

00079 delete [] \_\_ptr;

00080 }

00081 };

00082

00083 /// 20.7.12.2 unique\_ptr for single objects.

00084 template <typename \_Tp, typename \_Tp\_Deleter = default\_delete<\_Tp> >

[00085](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html) class [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)

00086 {

00087 typedef [std::tuple<\_Tp\*, \_Tp\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00550.html) [\_\_tuple\_type](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00550.html);

00088 typedef [\_\_tuple\_type](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00550.html) [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)::\* \_\_unspecified\_bool\_type;

00089 typedef \_Tp\* [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)::\* \_\_unspecified\_pointer\_type;

00090

00091 public:

00092 typedef \_Tp\* pointer;

00093 typedef \_Tp element\_type;

00094 typedef \_Tp\_Deleter deleter\_type;

00095

00096 // Constructors.

00097 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)()

00098 : \_M\_t(pointer(), deleter\_type())

00099 { static\_assert(![std::is\_pointer<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00414.html),

00100 "constructed with null function pointer deleter"); }

00101

00102 explicit

00103 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(pointer \_\_p)

00104 : \_M\_t(\_\_p, deleter\_type())

00105 { static\_assert(![std::is\_pointer<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00414.html),

00106 "constructed with null function pointer deleter"); }

00107

00108 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(pointer \_\_p,

00109 typename std::conditional<[std::is\_reference<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00416.html),

00110 deleter\_type, const deleter\_type&>::type \_\_d)

00111 : \_M\_t(\_\_p, \_\_d) { }

00112

00113 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(pointer \_\_p,

00114 typename std::remove\_reference<deleter\_type>::type&& \_\_d)

00115 : \_M\_t([std::move](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01025.html#g956195699e0833a97784b6111277f7e3)(\_\_p), [std::move](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01025.html#g956195699e0833a97784b6111277f7e3)(\_\_d))

00116 { static\_assert(![std::is\_reference<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00416.html),

00117 "rvalue deleter bound to reference"); }

00118

00119 // Move constructors.

00120 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)([unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00121 : \_M\_t(\_\_u.release(), std::forward<deleter\_type>(\_\_u.get\_deleter())) { }

00122

00123 template<typename \_Up, typename \_Up\_Deleter>

00124 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)([unique\_ptr<\_Up, \_Up\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00125 : \_M\_t(\_\_u.release(), std::forward<deleter\_type>(\_\_u.get\_deleter()))

00126 { }

00127

00128 // Destructor.

00129 ~[unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)() { reset(); }

00130

00131 // Assignment.

00132 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&

00133 operator=([unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00134 {

00135 reset(\_\_u.release());

00136 get\_deleter() = [std::move](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01025.html#g956195699e0833a97784b6111277f7e3)(\_\_u.get\_deleter());

00137 return \*this;

00138 }

00139

00140 template<typename \_Up, typename \_Up\_Deleter>

00141 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&

00142 operator=([unique\_ptr<\_Up, \_Up\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00143 {

00144 reset(\_\_u.release());

00145 get\_deleter() = [std::move](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01025.html#g956195699e0833a97784b6111277f7e3)(\_\_u.get\_deleter());

00146 return \*this;

00147 }

00148

00149 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&

00150 operator=(\_\_unspecified\_pointer\_type)

00151 {

00152 reset();

00153 return \*this;

00154 }

00155

00156 // Observers.

00157 typename std::add\_lvalue\_reference<element\_type>::type [operator\*](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01004.html#g02bdbe21b5a753599173b4e2c77b5497)() const

00158 {

00159 \_GLIBCXX\_DEBUG\_ASSERT(get() != 0);

00160 return \*get();

00161 }

00162

00163 pointer

00164 operator->() const

00165 {

00166 \_GLIBCXX\_DEBUG\_ASSERT(get() != 0);

00167 return get();

00168 }

00169

00170 pointer

00171 get() const

00172 { return std::get<0>(\_M\_t); }

00173

00174 typename std::add\_lvalue\_reference<deleter\_type>::type

00175 get\_deleter()

00176 { return std::get<1>(\_M\_t); }

00177

00178 typename [std::add\_lvalue\_reference](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00281.html)<

00179 typename [std::add\_const<deleter\_type>::type](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00279.html)

00180 >::type

00181 get\_deleter() const

00182 { return std::get<1>(\_M\_t); }

00183

00184 operator \_\_unspecified\_bool\_type () const

00185 { return get() == 0 ? 0 : &unique\_ptr::\_M\_t; }

00186

00187 // Modifiers.

00188 pointer

00189 release()

00190 {

00191 pointer \_\_p = get();

00192 std::get<0>(\_M\_t) = 0;

00193 return \_\_p;

00194 }

00195

00196 void

00197 reset(pointer \_\_p = pointer())

00198 {

00199 if (\_\_p != get())

00200 {

00201 get\_deleter()(get());

00202 std::get<0>(\_M\_t) = \_\_p;

00203 }

00204 }

00205

00206 void

00207 swap([unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00208 {

00209 using std::swap;

00210 swap(\_M\_t, \_\_u.\_M\_t);

00211 }

00212

00213 // Disable copy from lvalue.

00214 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(const [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&) = delete;

00215

00216 template<typename \_Up, typename \_Up\_Deleter>

00217 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(const [unique\_ptr<\_Up, \_Up\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&) = delete;

00218

00219 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)& operator=(const [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&) = delete;

00220

00221 template<typename \_Up, typename \_Up\_Deleter>

00222 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)& operator=(const [unique\_ptr<\_Up, \_Up\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&) = delete;

00223

00224 private:

00225 [\_\_tuple\_type](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00550.html) \_M\_t;

00226 };

00227

00228 /// 20.7.12.3 unique\_ptr for array objects with a runtime length

00229 // [unique.ptr.runtime]

00230 // \_GLIBCXX\_RESOLVE\_LIB\_DEFECTS

00231 // DR 740 - omit specialization for array objects with a compile time length

00232 template<typename \_Tp, typename \_Tp\_Deleter>

[00233](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00563.html) class [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)<\_Tp[], \_Tp\_Deleter>

00234 {

00235 typedef [std::tuple<\_Tp\*, \_Tp\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00550.html) [\_\_tuple\_type](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00550.html);

00236 typedef [\_\_tuple\_type](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00550.html) [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)::\* \_\_unspecified\_bool\_type;

00237 typedef \_Tp\* [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)::\* \_\_unspecified\_pointer\_type;

00238

00239 public:

00240 typedef \_Tp\* pointer;

00241 typedef \_Tp element\_type;

00242 typedef \_Tp\_Deleter deleter\_type;

00243

00244 // Constructors.

00245 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)()

00246 : \_M\_t(pointer(), deleter\_type())

00247 { static\_assert(![std::is\_pointer<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00414.html),

00248 "constructed with null function pointer deleter"); }

00249

00250 explicit

00251 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(pointer \_\_p)

00252 : \_M\_t(\_\_p, deleter\_type())

00253 { static\_assert(![std::is\_pointer<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00414.html),

00254 "constructed with null function pointer deleter"); }

00255

00256 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(pointer \_\_p,

00257 typename std::conditional<[std::is\_reference<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00416.html),

00258 deleter\_type, const deleter\_type&>::type \_\_d)

00259 : \_M\_t(\_\_p, \_\_d) { }

00260

00261 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(pointer \_\_p,

00262 typename std::remove\_reference<deleter\_type>::type && \_\_d)

00263 : \_M\_t([std::move](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01025.html#g956195699e0833a97784b6111277f7e3)(\_\_p), [std::move](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01025.html#g956195699e0833a97784b6111277f7e3)(\_\_d))

00264 { static\_assert(![std::is\_reference<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00416.html),

00265 "rvalue deleter bound to reference"); }

00266

00267 // Move constructors.

00268 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)([unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00269 : \_M\_t(\_\_u.release(), std::forward<deleter\_type>(\_\_u.get\_deleter())) { }

00270

00271 template<typename \_Up, typename \_Up\_Deleter>

00272 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)([unique\_ptr<\_Up, \_Up\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00273 : \_M\_t(\_\_u.release(), std::forward<deleter\_type>(\_\_u.get\_deleter()))

00274 { }

00275

00276 // Destructor.

00277 ~[unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)() { reset(); }

00278

00279 // Assignment.

00280 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&

00281 operator=([unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00282 {

00283 reset(\_\_u.release());

00284 get\_deleter() = [std::move](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01025.html#g956195699e0833a97784b6111277f7e3)(\_\_u.get\_deleter());

00285 return \*this;

00286 }

00287

00288 template<typename \_Up, typename \_Up\_Deleter>

00289 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&

00290 operator=([unique\_ptr<\_Up, \_Up\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00291 {

00292 reset(\_\_u.release());

00293 get\_deleter() = [std::move](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01025.html#g956195699e0833a97784b6111277f7e3)(\_\_u.get\_deleter());

00294 return \*this;

00295 }

00296

00297 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&

00298 operator=(\_\_unspecified\_pointer\_type)

00299 {

00300 reset();

00301 return \*this;

00302 }

00303

00304 // Observers.

00305 typename std::add\_lvalue\_reference<element\_type>::type

00306 operator[](size\_t \_\_i) const

00307 {

00308 \_GLIBCXX\_DEBUG\_ASSERT(get() != 0);

00309 return get()[\_\_i];

00310 }

00311

00312 pointer

00313 get() const

00314 { return std::get<0>(\_M\_t); }

00315

00316 typename std::add\_lvalue\_reference<deleter\_type>::type

00317 get\_deleter()

00318 { return std::get<1>(\_M\_t); }

00319

00320 typename [std::add\_lvalue\_reference](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00281.html)<

00321 typename [std::add\_const<deleter\_type>::type](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00279.html)

00322 >::type

00323 get\_deleter() const

00324 { return std::get<1>(\_M\_t); }

00325

00326 operator \_\_unspecified\_bool\_type () const

00327 { return get() == 0 ? 0 : &unique\_ptr::\_M\_t; }

00328

00329 // Modifiers.

00330 pointer

00331 release()

00332 {

00333 pointer \_\_p = get();

00334 std::get<0>(\_M\_t) = 0;

00335 return \_\_p;

00336 }

00337

00338 void

00339 reset(pointer \_\_p = pointer())

00340 {

00341 if (\_\_p != get())

00342 {

00343 get\_deleter()(get());

00344 std::get<0>(\_M\_t) = \_\_p;

00345 }

00346 }

00347

00348 // DR 821.

00349 template<typename \_Up>

00350 void reset(\_Up) = delete;

00351

00352 void

00353 swap([unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&& \_\_u)

00354 {

00355 using std::swap;

00356 swap(\_M\_t, \_\_u.\_M\_t);

00357 }

00358

00359 // Disable copy from lvalue.

00360 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(const [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&) = delete;

00361 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)& operator=(const [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)&) = delete;

00362

00363 // Disable construction from convertible pointer types.

00364 // (N2315 - 20.6.5.3.1)

00365 template<typename \_Up>

00366 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(\_Up\*, typename

00367 std::conditional<[std::is\_reference<deleter\_type>::value](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00416.html),

00368 deleter\_type, const deleter\_type&>::type,

00369 typename std::enable\_if<std::is\_convertible<\_Up\*,

00370 pointer>::value>::type\* = 0) = delete;

00371

00372 template<typename \_Up>

00373 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(\_Up\*, typename std::remove\_reference<deleter\_type>::type&&,

00374 typename std::enable\_if<std::is\_convertible<\_Up\*,

00375 pointer>::value>::type\* = 0) = delete;

00376

00377 template<typename \_Up>

00378 explicit

00379 [unique\_ptr](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)(\_Up\*, typename std::enable\_if<std::is\_convertible<\_Up\*,

00380 pointer>::value>::type\* = 0) = delete;

00381

00382 private:

00383 [\_\_tuple\_type](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00550.html) \_M\_t;

00384 };

00385

00386 template<typename \_Tp, typename \_Tp\_Deleter>

00387 inline void

00388 swap([unique\_ptr<\_Tp, \_Tp\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)& \_\_x,

00389 [unique\_ptr<\_Tp, \_Tp\_Deleter>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html)& \_\_y)

00390 { \_\_x.[swap](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html#d291b67ef05c7f74b62077cc6a20dc91)(\_\_y); }

00391

00392 template<typename \_Tp, typename \_Tp\_Deleter>

00393 inline void

00394 swap(unique\_ptr<\_Tp, \_Tp\_Deleter>&& \_\_x,

00395 unique\_ptr<\_Tp, \_Tp\_Deleter>& \_\_y)

00396 { \_\_x.[swap](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00562.html#d291b67ef05c7f74b62077cc6a20dc91)(\_\_y); }

00397

00398 template<typename \_Tp, typename \_Tp\_Deleter>

00399 inline void

00400 swap(unique\_ptr<\_Tp, \_Tp\_Deleter>& \_\_x,

00401 unique\_ptr<\_Tp, \_Tp\_Deleter>&& \_\_y)

00402 { \_\_x.swap(\_\_y); }

00403

00404 template<typename \_Tp, typename \_Tp\_Deleter,

00405 typename \_Up, typename \_Up\_Deleter>

00406 inline bool

00407 [operator==](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00967.html#8be0f873cd3c00ecedc6b075be3c7a1f)(const unique\_ptr<\_Tp, \_Tp\_Deleter>& \_\_x,

00408 const unique\_ptr<\_Up, \_Up\_Deleter>& \_\_y)

00409 { return \_\_x.get() == \_\_y.get(); }

00410

00411 template<typename \_Tp, typename \_Tp\_Deleter,

00412 typename \_Up, typename \_Up\_Deleter>

00413 inline bool

00414 [operator!=](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00967.html#59d315b3aa5503798a8f25150e864ad7)(const unique\_ptr<\_Tp, \_Tp\_Deleter>& \_\_x,

00415 const unique\_ptr<\_Up, \_Up\_Deleter>& \_\_y)

00416 { return !(\_\_x.get() == \_\_y.get()); }

00417

00418 template<typename \_Tp, typename \_Tp\_Deleter,

00419 typename \_Up, typename \_Up\_Deleter>

00420 inline bool

00421 operator<(const unique\_ptr<\_Tp, \_Tp\_Deleter>& \_\_x,

00422 const unique\_ptr<\_Up, \_Up\_Deleter>& \_\_y)

00423 { return \_\_x.get() < \_\_y.get(); }

00424

00425 template<typename \_Tp, typename \_Tp\_Deleter,

00426 typename \_Up, typename \_Up\_Deleter>

00427 inline bool

00428 operator<=(const unique\_ptr<\_Tp, \_Tp\_Deleter>& \_\_x,

00429 const unique\_ptr<\_Up, \_Up\_Deleter>& \_\_y)

00430 { return !(\_\_y.get() < \_\_x.get()); }

00431

00432 template<typename \_Tp, typename \_Tp\_Deleter,

00433 typename \_Up, typename \_Up\_Deleter>

00434 inline bool

00435 [operator>](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00967.html#5b64acaa6ccc3059f79b6589d35eee02)(const unique\_ptr<\_Tp, \_Tp\_Deleter>& \_\_x,

00436 const unique\_ptr<\_Up, \_Up\_Deleter>& \_\_y)

00437 { return \_\_y.get() < \_\_x.get(); }

00438

00439 template<typename \_Tp, typename \_Tp\_Deleter,

00440 typename \_Up, typename \_Up\_Deleter>

00441 inline bool

00442 [operator>=](https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00967.html#0ebdf8990492fc8bd7d485e5b333db62)(const unique\_ptr<\_Tp, \_Tp\_Deleter>& \_\_x,

00443 const unique\_ptr<\_Up, \_Up\_Deleter>& \_\_y)

00444 { return !(\_\_x.get() < \_\_y.get()); }

00445

00446 // @} group pointer\_abstractions

00447

00448 \_GLIBCXX\_END\_NAMESPACE

00449

00450 #endif /\* \_UNIQUE\_PTR\_H \*/

## smart\_ptr 구현

#ifndef SMART\_PTR

#define SMART\_PTR

template<typename T>

class MySmartPtr

{

public:

MySmartPtr(T\* \_ptr = nullptr) : ptr(\_ptr) {}

MySmartPtr(const MySmartPtr& \_ptr) = delete;

MySmartPtr(MySmartPtr&& \_ptr) : ptr(\_ptr.ptr)

{ \_ptr.ptr = nullptr; }

~MySmartPtr() { if(ptr) delete ptr; }

MySmartPtr& operator=(const MySmartPtr& \_ptr) = delete;

MySmartPtr& operator=(MySmartPtr&& \_ptr)

{

ptr = \_ptr.ptr;

\_ptr.ptr = nullptr;

}

T\* operator->() const { return ptr; }

T& operator\*() const { return \*ptr; }

//벙어리 포인터로 암시적 변환을 위한 연산자

//operator void\*() { return ptr; }

operator T\*() { return ptr; }

//상속된 클래스로의 암시적 변환을 위한 연산자

template<typename U>

operator MySmartPtr<U>()

{

MySmartPtr<U> ret(ptr);

ptr = nullptr;

return ret;

}

private:

T\* ptr;

};

#endif

위의 구현을 아래에서 사용해본다.

//오버로딩되어있다. 인자는 필요하다면 해당 매개변수로 암시적 변환이 생긴다.

//인자로 blademaster, archmage도 가능하다.

void battle(const MySmartPtr<Archer>& target)

{

target->attack();

}

//오버로딩시, 자신과 가까운 함수를 찾는다.

//고로 blademaster은 변환이 필요한 위 함수 대신 아래 함수를 호출한다.

void battle(const MySmartPtr<BladeMaster>& target)

{

std::cout << "only sword\n";

target->attack();

}

int main()

{

int\* temp = new int(9999);

MySmartPtr<int> ptr(temp);

//이동 시, 소유권을 이전한다. 복사는 금지된다.

//MySmartPtr<int> ptr2(ptr);

MySmartPtr<int> ptr2(std::move(ptr));

//암시적 변환자를 통해 벙어리 포인터처럼 사용하기

if (temp) std::cout << \*temp << std::endl;

if (ptr) std::cout << \*ptr << std::endl;

if (ptr2) std::cout << \*ptr2 << std::endl;

std::unique\_ptr<int> uptr = std::make\_unique<int>(8888);

if(uptr) std::cout << \*uptr << std::endl;

if(uptr == std::make\_unique<int>(8888))

std::cout << \*uptr << std::endl;

MySmartPtr<BladeMaster> bmptr(new BladeMaster);

battle(bmptr);

return 0;

}

# [항목29] 참조 카운팅

## 참조 카운팅 기능을 가진 String구현

class String

{

public:

class LengthOverflowException : public std::exception

{

public:

virtual const char\* what() const

{ return "예외: LengthOverflowException\n"; };

};

String(const char\* newValue = "")

: value(new RefCounter(newValue)) {}

~String()

{

if (--value->refCount == 0)

{ delete value; }

}

String(const String& rhs)

{

if (rhs.value->sharable)

{

value = rhs.value;

++value->refCount;

}

else

{

value = new RefCounter(rhs.value->data);

}

}

String& operator=(const String& rhs)

{

if (value == rhs.value) return \*this;

value = rhs.value;

++value->refCount;

return \*this;

}

const char& operator[](int index) const

{

if (index < 0 || index >= strlen(value->data))

{

throw LengthOverflowException();

}

return value->data[index];

}

//refCounter로 인하여 쓰기 동작과 읽기 동작을 구분해야 한다.

char& operator[](int index)

{

if (value->refCount > 1)

{

--value->refCount;

value = new RefCounter(value->data);

}

value->sharable = false;

return const\_cast<char&>(

static\_cast<const String&>(\*this)[index]

);

}

private:

struct RefCounter

{

RefCounter(const char\* newValue) : refCount(1), sharable(true)

{

data = new char[strlen(newValue) + 1];

strcpy(data, newValue);

std::cout << "create String.\n";

}

~RefCounter() { delete[] data; }

int refCount;

bool sharable;

char\* data;

} \*value;

};

같은 문자열을 공유하여 사용할 수 있다.

## 참조 카운팅 클래스

이러한 기능을 객체로 분리하여 어떤 자료이든 참조 카운팅을 할 수 있도록 만들어야 한다.

#pragma once

#ifndef RCPTR

#define RCPTR

template<typename T>

class RCPtr

{

public:

RCPtr(T\* realPtr = nullptr);

RCPtr(const RCPtr& rhs);

~RCPtr();

RCPtr<T>& operator=(const RCPtr& rhs);

T\* operator->() const;

T& operator\*() const;

private:

void init();

private:

T\* pointee;

};

template<typename T>

RCPtr<T>::RCPtr(T \* realPtr)

: pointee(realPtr)

{

init();

}

template<typename T>

RCPtr<T>::RCPtr(const RCPtr & rhs)

: pointee(rhs.pointee)

{

init();

}

template<typename T>

RCPtr<T>::~RCPtr()

{

if (pointee) pointee->removeReference();

}

template<typename T>

RCPtr<T> & RCPtr<T>::operator=(const RCPtr & rhs)

{

if (pointee != rhs.pointee)

{

T\* oldPointee = pointee;

pointee = rhs.pointee;

init();

if (oldPointee)

{

oldPointee->removeReference();

}

}

// TODO: 여기에 반환 구문을 삽입합니다.

return \*this;

}

template<typename T>

T \* RCPtr<T>::operator->() const

{

return pointee;

}

template<typename T>

T & RCPtr<T>::operator\*() const

{

// TODO: 여기에 반환 구문을 삽입합니다.

return \*pointee;

}

template<typename T>

void RCPtr<T>::init()

{

if (pointee == nullptr)

{

return;

}

if (pointee->isSharable() == false)

{

pointee = new T(\*pointee);

}

pointee->addReference();

}

//#include "RCPtr.inl";

#endif

#pragma once

#ifndef RCOBJECT

#define RCOBJECT

class RCObject

{

public:

RCObject();

RCObject(const RCObject& rhs);

RCObject& operator=(const RCObject& rhs);

virtual ~RCObject() = 0;

void addReference();

void removeReference();

void markUnsharable();

bool isSharable() const;

bool isShared() const;

private:

int refCount;

bool sharable;

};

RCObject::RCObject()

: refCount(0), sharable(true)

{

}

RCObject::RCObject(const RCObject & rhs)

: refCount(0), sharable(true)

{

}

RCObject & RCObject::operator=(const RCObject & rhs)

{

// TODO: 여기에 반환 구문을 삽입합니다.

return \*this;

}

RCObject::~RCObject()

{

}

void RCObject::addReference()

{

++refCount;

}

void RCObject::removeReference()

{

if (--refCount == 0) delete this;

}

void RCObject::markUnsharable()

{

sharable = false;

}

bool RCObject::isSharable() const

{

return sharable;

}

bool RCObject::isShared() const

{

return refCount > 1;

}

#endif // !

#include "RCObject.h"

#include "RCPtr.h"

class String

{

public:

class LengthOverflowException : public std::exception

{

public:

virtual const char\* what() const

{ return "예외: LengthOverflowException\n"; };

};

String(const char\* newValue = "")

: value(new StringValue(newValue)) {}

const char& operator[](int index) const

{

if (index < 0 || index >= strlen(value->data))

{

throw LengthOverflowException();

}

return value->data[index];

}

//refCounter로 인하여 쓰기 동작과 읽기 동작을 구분해야 한다.

char& operator[](int index)

{

if (value->isShared())

{

value->removeReference();

value = new StringValue(value->data);

}

value->markUnsharable();

return const\_cast<char&>(

static\_cast<const String&>(\*this)[index]

);

}

private:

struct StringValue : public RCObject

{

StringValue(const char\* newValue)

{

init(newValue);

std::cout << "create new string value.\n";

}

StringValue(const StringValue& rhs)

{

init(rhs.data);

}

~StringValue() { delete[] data; }

void init(const char\* newValue)

{

data = new char[strlen(newValue) + 1];

strcpy(data, newValue);

}

char\* data;

};

RCPtr<StringValue> value;

};

위의 구조는 다음과 같다.

String [has RCPtr]   
RCPtr -> StringValue [상속: RCObject]   
StringValue -> char\* data

## 부착 가능한 참조 카운팅 클래스

위와 같은 방식은 사용하고자 하는 클래스 안에 포인터가 참조 카운팅 기능을 수행하기 때문에 이미 존재하는 객체에는 카운팅 기능을 추가하기 어렵다.

이미 존재하는 객체에 카운팅을 추가할 수 있는 객체를 구현해야 한다.

#ifndef RCIPTR

#define RCIPTR

#include "RCObject.h"

template<typename T>

class RCIPtr

{

public:

RCIPtr(T\* realPtr = nullptr);

RCIPtr(const RCIPtr& rhs);

~RCIPtr();

RCIPtr<T>& operator=(const RCIPtr& rhs);

const T\* operator->() const;

T\* operator->() const;

const T& operator\*() const;

T& operator\*() const;

private:

void init();

void makeCopy();

private:

struct CountHolder : public RCObject

{

~CountHolder() { delete pointee; }

T\* pointee;

};

CountHolder\* counter;

};

template<typename T>

RCIPtr<T>::RCIPtr(T \* realPtr)

: counter(new CountHolder)

{

counter->pointee = realPtr;

init();

}

template<typename T>

RCIPtr<T>::RCIPtr(const RCIPtr & rhs)

: counter(rhs.counter)

{

init();

}

template<typename T>

RCIPtr<T>::~RCIPtr()

{

counter->removeReference();

}

template<typename T>

RCIPtr<T> & RCIPtr<T>::operator=(const RCIPtr & rhs)

{

if (counter != rhs.counter)

{

counter->removeReference();

counter = rhs.counter;

init();

}

// TODO: 여기에 반환 구문을 삽입합니다.

return \*this;

}

template<typename T>

T \* RCIPtr<T>::operator->() const

{

makeCopy();

return counter->pointee;

}

template<typename T>

T & RCIPtr<T>::operator\*() const

{

makeCopy();

// TODO: 여기에 반환 구문을 삽입합니다.

return \*(counter->pointee);

}

template<typename T>

const T \* RCIPtr<T>::operator->() const

{

return counter->pointee;

}

template<typename T>

const T & RCIPtr<T>::operator\*() const

{

// TODO: 여기에 반환 구문을 삽입합니다.

return \*(counter->pointee);

}

template<typename T>

void RCIPtr<T>::init()

{

if (counter->isSharable() == false)

{

T\* oldPointee = counter->pointee;

counter = new CountHolder;

counter->pointee = new T(\*oldPointee);

}

counter->addReference();

}

template<typename T>

void RCIPtr<T>::makeCopy()

{

if (counter->isShared())

{

T\* oldPointee = counter->pointee;

counter->removeReference();

counter = new CountHolder;

counter->pointee = new T(\*oldPointee);

counter->addReference();

}

}

//#include "RCPtr.inl";

#endif

RCIPtr클래스는 원본 포인터, 카운팅 기능을 모두 포함하고 있다는 사실을 알 수 있다. 템플릿 인자에 원하는 클래스만 넣으면 RCIPtr은 해당 클래스를 카운팅할 수 있는 포인터를 갖게 되는 것이다.

# [항목30] 프록시 클래스

프록시 클래스는 어떤 객체를 대신하여 동작하게 하는 장치이다. 그 동작의 종류는 다양하다.

## 2차원 배열

template<typename T>

class Array2D

{

private:

//proxy class

class Array1D

{

public:

Array1D() : data(nullptr) {}

Array1D(int s) : data(new T[s]) {}

~Array1D()

{

if (data) delete[] data;

}

T& operator[](int idx) { return data[idx]; }

const T& operator[](int idx) const{ return data[idx]; }

void resize(int s)

{

if (data == nullptr)

data = new T[s];

}

private:

T\* data;

};

public:

Array2D(int s1, int s2)

{

data = new Array1D[s1];

for (int i = 0; i < s1; i++)

data[i].resize(s2);

}

~Array2D()

{

if (data) delete[] data;

}

Array1D& operator[](int idx) { return data[idx]; }

const Array1D& operator[](int idx) const{ data[idx]; }

private:

Array1D\* data;

};

int main()

{

Array2D<float> arr(10, 20);

arr[3][4] = 7;

std::cout << arr[3][4] << std::endl;

return 0;

}

내부에서 1차원 배열의 보조 클래스를 이용하여 구현한다. 이로써 연산자[][]를 중복하여 사용할 수 있게 한다. 원리는 객체를 반환한다는 것이다. 그 객체는 뭔가의 기능을 덧붙일 수 있고 더 다양한 기능을 할 수 있게 한다.

## 프록시 클래스의 예시

**class** **UndoString** : **public** std::string

{

**struct** proxy

{

UndoString \* str;

size\_t pos;

proxy(UndoString \* us, size\_t position)

: str(us), pos(position)

{}

*// Invoked when proxy is used to modify the value.*

void **operator** = (**const** char & rhs)

{

str->old = str->at(pos);

str->old\_pos = pos;

str->at(pos) = rhs;

}

*// Invoked when proxy is used to read the value.*

**operator** **const** char & () **const**

{

**return** str->at(pos);

}

};

char old;

int old\_pos;

**public**:

UndoString(**const** std::string & s)

: std::string(s), old(0), old\_pos(-1)

{}

*// This operator replaces std::string's non-const operator [].*

proxy **operator** [] (size\_t index)

{

**return** proxy(**this**, index);

}

**using** std::string::**operator** [];

void undo()

{

**if**(old\_pos == -1)

**throw** std::runtime\_error("Nothing to undo!");

std::string::at(old\_pos) = old;

old = 0;

old\_pos = -1;

}

};

## 특징

어떤 타입을 대신하는 것이다. 그러므로 대신하려면 기존의 타입의 기능을 가져야 할 수도 있다. 기본적인 연산인 +, -등이 그것이다.

암시적 변환에 대해 고려하여 오버로딩 되어있다. 하지만 2중으로 암시적 변환이 생긴다면 작동하지 않는다.