TIME TABLE

구 분	오 전	오 후
1일차	● C와 다른 C++언어의 특징 - Namespace - C++ 입출력 - C++ 함수의 특징 - C++ 변수의 특징 - Auto, 새로운 제어문등	● C와 다른 C++ 언어의 문법 - new - reference - Casting 등
2일차	● 클래스 문법 1 - 객체지향 프로그래밍의 개념 - 생성자, 소멸자 - 초기화 리스트	● 클래스 문법 - Static 멤버 - Const 멤버 - this 포인터
3일차	● 상속 - upcast과 가상함수 - Coupling	● 추상 클래스와 인터페이스 - 추상 클래스 개념/활용 - 인터페이스 기반 설계
4일차	● 연산자 재정의 - 연산자 재정의 개념 - 주요 연산자 재정의 기술 - 함수객체, 스마트 포인터, cout의 원리등	● 객체의 복사 - 복사 생성자 개념 - 얕은 복사 vs 깊은 복사 - 참조계수
5일차	● 예외 처리 - 반환값 vs 예외 처리 차이점 - C++ 표준 예외	● STL 라이브러리 소개/활용 - container 활용 - Algorithm 활용

현대자동차 SW Intensive Program 교재

목 차

SECTION 1

C++ Better Than C

C++은 C 언어에 "객체지향 프로그래밍" 과 "일반화 프로그래밍"의 개념을 추가한 언어 입니다.

하지만, 위 2 가지 요소 뿐 아니라 C++은,

① 기존의 C 언어가 가지고 있던 변수, 함수, 제어문과 반복문, 메모리 할당, 포인터 등의 개념을 좀더 강력하게 사용할 수 있도록 다양한 문법을 추가 했습니다.

② 또한, **C++11/14/17** 에서 기존의 C++에 다양한 요소를 추가 했습니다.

이번 장에서는 본격적인 "객체지향 프로그래밍", "일반화 프로그래밍"을 다루기 전에 C 언어 보다 좋아진 C++의 다양한 기본 요소를 살펴 보도록 하겠습니다.

이번 장에서 다음과 같은 개념을 배우게 됩니다.

"C++의 역사, namespace, 표준입출력, 타입과 변수, 새로운 제어문과 반복문, C++ 함수의 특징, 동적 메모리 할당, reference, C++ explicit casting"

About C++

C++의 역사

- C++의 탄생

C++은 1979 년경에 " C with Classes" 라는 이름으로 "Bjarne Stroustrup"에 의해 탄생되었습니다.

- C++98/03

C++ 탄생 이후 표준화 없이 계속 사용되다가 1990 년 C++ 표준위원회가 설립되고, 1998 년 드디어 1 차 표준화를 하게 됩니다. 이때 표준화된 문법을 "C++98" 이라고 합니다. C++ 표준 라이브러리인 STL(Standard Template Library)도 이 때 나오게 됩니다. 그후, 2003 년에 표준화된 문법의 사소한 버그를 수정하고 몇가지를 추가해서 새로운 표준을 발표 합니다. 이 버전을 "C++03"이라고 합니다. 또는, 2 개 버전을 합쳐서 "C++98/03" 이라고 부르기도 합니다.

- C++11/14 (C++0x, C++1y)

C++93/03 이후 2000 년대에 들어 C++ 표준위원회는 새로운 표준을 만들기로 합니다. 새로운 표준을 2009 년 전에는 표준을 완성하겠다는 의지를 가지고 한자리 연도를 의미하는 "C++0x" 라는 이름으로 부르게 됩니다. 하지만, 의지와는 다르게 2 자리수의 연도인 2011 년에 표준이 완성되고 ISO 를 통과 하게 됩니다. 이 새로운 문법을 "C++11" 이라고 합니다. 그 후, 표준화 내용의 사소한 버그를 수정하고 몇가지 기능을 추가해서 2014 년 "C++14"를 발표 합니다. "C++14"는 발표 되기 전까지 "C++1y" 라는 이름을 사용했습니다. C++11 이 major update 였다면 C++14 는 minor update 입니다. 둘을 합쳐서 흔히 "C++11/14" 라고 합니다.

- C++17/20 (C++1z)

C++11/14 발표 이후 다시 표준위원회는 새로운 표준을 만들기 위해 노력 하게 됩니다. C++11/14 발표후 C++ 표준위원회는 즉시 "C++1z"라는 프로젝트 이름으로 차기 버전의 표준화를 준비합니다. "C++1z"은 2017 년에 "C++17" 이라는 이름으로 발표 되었습니다. 그리고 2020 년에 다시 minor update 를 포함한 C++20 을 발표할 예정입니다. 현재 C++ 표준위원회는 3 년을 주기로 새로운 버전을 발표하려고 노력하고 있습니다.

이 책에서는

"C++11/14/17 까지를 이야기 합니다."

비록, C++14 가 2014 년에 발표 되지만, C++14 를 완벽히 지원하는 컴파일러가 나오고 개발 현장에서 사용되기 까지는 몇년 정도의 시간이 걸리게 되었고, C++17 역시, 2017 년에 표준화를 완성하고 발표 했지만, C++17 문법을 완전히 지원하는 컴파일러가 나오고, 현업에서 널리 사용되기 까지는 어느 정도의 시간이 필요합니다.

이 책에서는 C++11/14/17 까지의 내용을 다루고 있습니다. 또한, C++11/14/17 에서 추가된 문법을 설명할 때는 제목 옆에 C++11/14/17 라고 표기 했습니다.

"C++의 각 주제를 깊이 있게 다루지는 않습니다."

이 책에서는 C++의 각 주제를 아주 깊이 있기 다루지는 않습니다. C++은 너무 방대해서 처음 시작하는 사람이 한권의 책으로 모든 내용을 깊이 있게 이해 할 수는 없습니다. 각 주제의 깊이 있는 내용은 "C++ 중급" 과정을 참고 하시기 바랍니다.

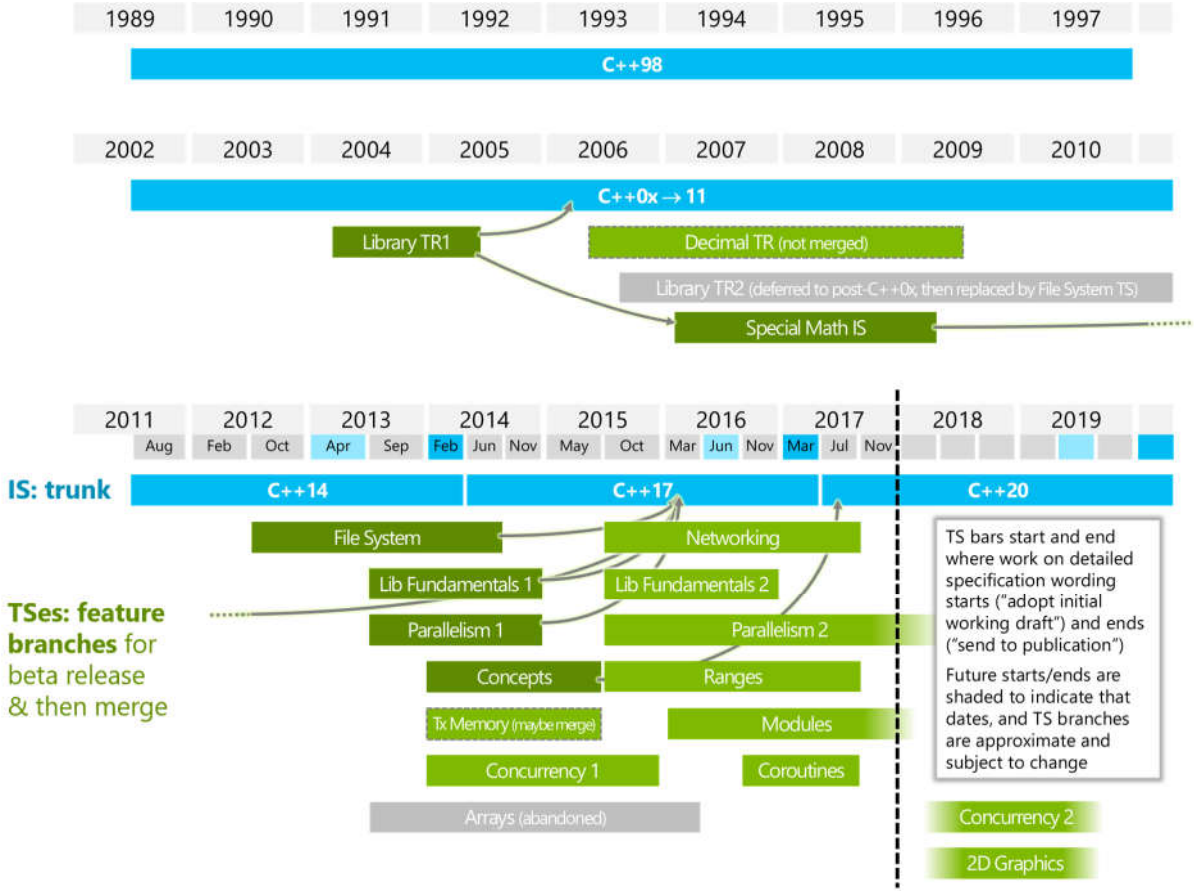
"C++의 모든 내용을 다루지는 않습니다."

C++11/14 의 핵심 기술이라고 할 수 있는 "Perfect forwarding", "Move Semantics" 등은 좀 복잡하고 어려운 내용이 많으므로 "C++ Intermediate" 과정에서 배우게 됩니다.

"STL 의 사용법을 간단히 배우게 됩니다."

C++ 표준 라이브러리인 STL 에 대해서는 간단한 개념과 사용법을 다루고 있습니다. STL 의 깊이 있는 내용이나 thread/Concurrency/chrono/smart pointer 등의 깊이 있는 내용은 "C++ STL Programming" 과정을 참고 하시기 바랍니다.

C++ 표준 위원회가 운영하는 isocpp.org 에서 다음의 그림과 같은 C++의 과거/현재/미래를 확인 할수 있습니다. (isocpp.org 사이트의 좌측 메뉴중 "Current ISO C++ Status" 메뉴)


1989
1990
1991
1992
1993
1994
1995
1996
1997
C++98
2002
2003
2004
2005
2006
2007
2008
2009
2010
C++0x → 11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS
2011
2012
2013
2014
2015
2016
2017
2018
2019
Aug
Feb
Oct
Apr
Sep
Feb
Jun
Nov
May
Oct
Mar
Jun
Nov
Mar
Jul
Nov
IS: trunk
C++14
C++17
C++20
File System
Networking
Lib Fundamentals 1
Lib Fundamentals 2
TSes: feature branches for beta release & then merge
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (maybe merge)
Modules
Concurrency 1
Coroutines
Arrays (abandoned)
TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send to publication")
Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change
Concurrency 2
2D Graphics

C++ 컴파일러

C++ 최신 문법을 사용하려면 되도록 최신의 컴파일러를 사용해야 합니다. 컴파일러별 문법 지원 현황을 보려면 아래 사이트를 참고 하면 됩니다.

http://en.cppreference.com/w/cpp/compiler_support

MS 의 VC++ 컴파일러(cl.exe)의 경우 C++11/14 문법을 사용하려면 VC++2015 이상 C++17/20 등의 최신 문법을 사용하려면 VC++2017 이상의 최신 버전을 사용하는 것이 좋습니다. 또한, VC++을 사용해서 C++17/20 등의 최신의 문법을 컴파일 하려면 **"/std:c++latest"** 옵션을 지정해야 합니다.

g++의 경우도 되도록 최신 버전을 사용하는 것이 좋습니다. 또한 g++의 경우 최신 문법을 사용하려면 -std 옵션을 지정해야 합니다

C++11 문법	`g++ -std=c++11 hello.cpp`
C++14 문법	`g++ -std=c++14 hello.cpp` 또는 `g++ -std=c++1y hello.cpp`
C++17 문법	`g++ -std=c++17 hello.cpp` 또는 `g++ -std=c++1z hello.cpp`

소스 파일 확장자

다양한 컴퓨터 언어들은 소스 코드를 만들 때 자신 만의 확장자를 사용 합니다. C 언어는 .c, java 는 .java, C#은 .cs, 스위프트는 .swift 등을 사용합니다.

C++ 언어의 경우 하나의 확장자만 있는 것은 아니고, 컴파일러에 따라 다양한 형태의 확장자를 사용해 왔습니다.

compiler	File extension
Unix uses	.C, .cc, .cxx, .c
GNU C++ uses	.C, .cc, .cxx, .cpp, .c++
Digital Mars uses	.cpp, .cxx
Borland C++ uses	.cpp
Watcom uses	.cpp
Microsoft Visual C++ uses	.cpp, .cxx, .cc
Metrowerks CodeWarrior uses	.cpp, .cp, .cc, .cxx, .c++

예전부터 .cc, .cxx 등 다양한 확장자를 사용해 왔지만 요즘은 **.cpp** 확장자를 가장 널리 사용합니다. 본 교재 에서도 .cpp 확장자를 사용하도록 하겠습니다.

헤더 파일의 경우는 .h, .hpp 등을 많이 사용하지만, C++표준 자체의 헤더파일은 확장자를 사용하지 않습니다. 특히, 기존에 C 에서 사용하던 모든 헤더 파일을 C++에서도 사용할 수 있는데, C++에서는 기존 C 헤더의 확장자인 .h 를 제거 하고 헤더 파일 이름 앞에 "c"를 추가 했습니다.

```
#include <iostream> // C++ 표준 헤더, 헤더 파일의 확장자가 없습니다.

#include <stdio.h>  // C style 헤더, C++에서도 계속 이렇게 사용할 수 있습니다.
#include <cstdio>   // 하지만, C++에서는 앞에 c를 붙이고 .h를 제거한 형태를 권장합니다.
```

[참고] <stdio.h> 와 <cstdio> 의 차이점은 뒷장의 namespace 를 배울 때 다루게 됩니다.

1장. namespace

1.1 기본 개념

프로젝트의 규모가 커지면서 여러 명의 개발자가 수백~수천개의 파일로 프로그램을 작성하면 코드 관리가 어려워 지고, 함수의 이름이 충돌도 나올 수 있습니다.

C++언어의 namespace 문법을 사용하면

① 프로그램의 각 요소를 관련된 요소끼리 묶을 수도 있고

② 이름 충돌을 막을 수도 있습니다.

다음 코드는 Audio 관련 코드, Video 관련 코드를 각각의 namespace 로 관리하는 코드입니다. namespace 가 다르므로 동일한 모양의 함수 init() 을 사용할 수 있습니다.

```
namespace Audio
{
    void init() {}
}
namespace Video
{
    void init() {}
}

void init() {}

int main()
{
    init();         // global namespace 안에 있는 init 함수를 호출합니다.
    Audio::init();  // Audio 이름 공간 안에 있는 init 함수를 호출합니다.
    Video::init();  // Video 이름 공간 안에 있는 init 함수를 호출합니다.
}
```

특정한 namespace 에 속하지 않은 것을 "전역 이름 공간(global namespace)"라고 합니다.

namespace 를 만드는 방법은 다음과 같습니다.

```
namespace 이름
```

```
{
    // … …
    // namespace 안에 포함할 함수, 구조체등.

}
```

1.2 namespace 사용하기

namespace 안에 있는 함수, 구조체 등에 접근하는 방법은 3 가지가 있습니다.

① **전체 이름**(qualified name lookup)**을 사용한 방법**

② **using 선언(declaration)을 사용한 방법**

③ **using 지시어(directive)를 사용한 방법.**

```
Audio::init();          // 1. 완전한 이름을 사용한 호출

using Audio::init;      // 2. using 선언(Declaration)
init();                 //    init함수는 namespace 이름없이 사용 가능 합니다.

using namespace Audio;  // 3. using 지시어(Directive)
init();                 //    Audio namespace 에 있는 모든 요소를
                        //    Audio 이름없이 사용가능합니다.
```

위 코드에서 using 선언과 using 지시어의 차이점을 정확히 알아 두시기 바랍니다. 또한, using 선언과 using 지시어는 함수 안에 만들어도 되고, 함수 밖에 만들어도 됩니다.

using 선언 또는 지시어를 함수 안에서 만들 경우 해당 함수 안에서만 namespace 이름 없이 사용할 수 있습니다. 하지만 함수 밖에서 만들 경우 모든 함수에서 namespace 이름 없이 사용할 있습니다.

1.3 std namespace

C++ 표준에 있는 모든 요소는 "std" 라는 namespace 안에 제공하고 있습니다. C++ 표준 함수인 min 함수를 사용하려면 다음과 같은 3 가지 방법을 사용할수 있습니다.

```
// 완전한 이름을 사용
#include <algorithm>

int main()
{
    int n = std::min(1,2);
}
```

```
// using declaration
#include <algorithm>
using std::min;

int main()
{
    int n = min(1,2);
}
```

```
// using directive
#include <algorithm>
using namespace std;

int main()
{
    int n = min(1,2);
}
```

이중 3 번째 방법으로 하면 코드의 양을 줄일수 있지만 사용자가 만든 코드에서 사용한 이름이 std namespace 안에 있으면 error 가 발생할수 있습니다.

아래 코드는 사용자가 count 라는 이름의 전역변수를 만들어서 사용하고 있습니다. 아무 문제 없어 보이는 코드이지만 std namespace 안에 C++ 표준 함수인 count 함수가 있기 때문에 이름 충돌이 발생하게 됩니다.

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

int count = 0;
int main()
{
    count = 10;     // error. 사용자가 만든 count 와 std::count() 의 이름 충돌이
                    // 발생합니다.
}
```

이런 문제 때문에 되도록이면 완전한 이름("std::min") 으로 사용하는 것이 좋습니다.

[참고] C++ 표준함수등을 사용할 때 std:: 를 붙여서 사용하는 것이 좋지만 이 책의 일부 예제는 지면이 좁은 관계상 "using namespace std" 를 사용하는 경우도 있습니다.

1.4 C++ 헤더 파일

<stdio.h> 안에 있는 printf()는 std namespace 가 아닌 전역 이름공간에 만들어져 있습니다. 따라서, std::printf()처럼 사용하면 에러가 발생합니다.

```
#include <stdio.h>

int main()
{
    printf("hello\n");      // ok..
    std::printf("hello\n"); // error
}
```

하지만, C++ 에서는 모든 표준 라이브러리는 "std" 이름 공간 안에 넣고 싶었습니다. 그래서, C++언어에서는 <stdio.h> 헤더와는 별개로 <cstdio> 라는 헤더 파일을 추가로 만들었습니다.

<cstdio> 파일의 구현 원리는 컴파일러 마다 다를수 있지만 보통 다음과 같이 만들어 져 있습니다.

```
// cstdio 파일
#include <stdio.h>  // 1. 먼저 기존의 stdio.h 헤더를 포함 합니다.
                    //    printf 함수가 전역공간에 선언 됩니다.
namespace std
{
    // printf 함수(그리고 모든 C함수)를 std 안에도 선언을 포함해 줍니다.
    using ::printf;
    //......
}
```

따라서, <stdio.h> 파일 대신에 <cstdio> 파일을 포함하면 printf 등의 C 표준 함수를 전역공간에 있는 함수 처럼 사용할 수 도 있고 std namespace 이름을 붙여서도 사용할 수 있습니다.

```
#include <cstdio>

int main()
{
    printf("hello\n");      // ok
    std::printf("hello\n"); // ok
}
```

위와 같은 이유로 C++ 에서는 기존에 C 언어에 있던 헤더 파일을 사용할 때 헤더 파일의 이름 앞에 'c' 를 붙이고 '.h' 를 제거한 형태로 사용하게 됩니다.

<stdio.h> 대신 <cstdio> 를, <stdlib.h> 대신 <cstdlib>를, <string.h> 대신 <cstring> 를 사용합니다.

물론, C 언어와 같이 <stdio.h> 를 사용해도 문제가 되지는 않습니다.

2 장. C+ 표준 입출력

2.1 표준 입출력 기본

C 언어에서의 표준 입출력은 printf 함수와 scanf 함수를 사용합니다. 다음 코드는 정수 하나를 입력 받아 화면에 출력하는 C 언어로 만든 코드 입니다.

```c
#include <stdio.h>

int main()
{
    int n = 0;
    scanf("%d", &n);    // C 표준 입력
    printf("%d\n", n);  // C 표준 출력
    return 0;
}
```

C++ 언어는 C 언어의 모든 것을 지원하므로 printf, scanf 등을 계속 사용해도 되지만, C++은 자신만의 표준 입출력 방식도 제공 합니다.

다음 코드는 위와 동일한 일을 하는 C++ 코드 입니다.

```cpp
#include <iostream>

int main()
{
    int n = 0;
    std::cin >> n;      // C++ 표준 입력
    std::cout << n;     // C++ 표준 출력
    std::cout << std::endl; // endl은 개행('\n')을 할 때 사용합니다.
    return 0;
}
```

한줄씩 자세히 살펴 보도록 하겠습니다.

- Header file

```cpp
#include <iostream>
```

C 언어에서 printf 를 사용하려면 <stdio.h> 헤더를 포함해야 하듯이 C++표준 입출력 도구인 std::cout, std::cin 을 사용하려면 <iostream> 헤더를 포함해야 합니다.

일반적으로 C 언어에서는 헤더 파일이 .h 확장자를 사용하지만 C++에서는 헤더 파일에 확장자를 사용하지 않습니다. 또한, 기존의 C 에서 사용하던 헤더파일도 ".h" 확장자를 제거하고 파일 이름 앞에 "c" 를 붙여서 사용합니다.

```
#include <stdio.h>  // C 언어 헤더
#include <cstdio>   // C++ 언어 헤더
```

물론, C++에서 <stdio.h>를 사용해도 되지만 <cstdio>를 사용하는 것이 관례입니다. 두, 파일의 차이점에 대해서는 2 장의 namespace 를 참고하시면 됩니다.

- **표준 입력 std::cin**

```
std::cin >> n;          // C++ 표준 입력
```

C++에서 표준 입력은 cin 을 사용합니다. cin 은 std namespace 안에 있으므로 std::cin 으로 사용하면 됩니다.

사용법은 "std::cin >> 변수이름" 의 형식을 사용하게 됩니다.

C 언어의 scanf 와의 차이점은 **"%d", "%f" 등의 입력 데이터의 종류를 지정할 필요가 없고, 변수의 주소를 전달하지 않고 변수의 이름만** 적으면 됩니다.

```
scanf("%d", &n);// 정수 입력시 "%d"가 적어야 합니다. 변수의 주소(&n)을 전달해야 합니다.
std::cin >> n;      // 입력 data의 형태를 지정할 필요가 없습니다. 주소를 전달하지
않습니다
```

- **표준 출력 std::cout**

```
std::cout << n;     // C++ 표준 출력
std::cout << std::endl; // endl은 개행('\n')을 할 때 사용합니다.
```

C++에서 표준 출력은 std::cout 을 사용합니다. 사용법은 "std::cout << 변수이름" 의 형식을 사용하게 됩니다.

std::cin 과 마찬가지로 **"%d", "%f" 등의 입력 데이터의 종류를 지정할 필요가 없습니다.** 또한, 개행을 위해 C 처럼 '\n' 를 사용해도 되지만 보통은 std::endl 을 사용합니다.

```
printf("%d\n", n);  // 출력 data의 종류를 지정해야 합니다. '\n' 로 개행 합니다.
std::cout << n;         / 출력 data의 종류를 지정할 필요 없습니다.
std::cout << '\n';      // std::endl 대신 '\n'를 사용해도 됩니다.
```

std::cout 을 사용해서 여러 개의 변수를 출력 하기 위해 아래처럼 연속적으로 "<<" 를 사용 할수 도 있습니다.

```
std::cout << n1 << n2 << n3 << std::endl;   // 3개의 변수를 출력하고 개행을 출력합니다.
```

앞으로는 변수 값을 출력 하고 개행을 할 때 위 코드 처럼 한 줄 로 사용하도록 하겠습니다.

다음 단계로 나가기전에 다음에 대해서 생각해 보세요.

① **printf 와 scanf 는 뒤에 ()를 붙여 사용하므로 함수 입니다. 그런데, std::cout 과 std::cin 을 ()가 붙지 않으므로 함수는 아닙니다. 그렇다면, std::cout 과 std::cin 의 정체는 무엇일까요 ?**

② **C 언어에서 변수에 입력 값을 담아 오려면 주소를 보내야 합니다. 그런데, std::cin 은 주소를 사용하지 않습니다. 원리가 뭘까요 ?**

③ **C 언어는 입출력시에 "%d", "%f" 등을 사용해서 data 의 종류를 지정해야 합니다. 하지만, std::cin 은 종류를 지정할 필요가 없습니다. 원리가 뭘까요 ?**

④ **개행을 할 때 사용하는 std::endl 의 정체는 무엇일까요 ?**

위 질문에 대한 답은 좀 어려운 내용이므로 앞으로 C++의 다양한 문법을 배우면서 하나씩 해답을 찾아 보도록 하겠습니다. cout, cin 은 생각보다 어려운 내용이 많이 있습니다. 지금은 시작 단계 이므로 cout, cin 의 사용법만 정확히 알아 두면 충분합니다.

2.2 namespace 와 cout, cin

모든 C++ 표준 함수가 std namespace 안에 있는 것처럼 cout 과 cin 도 모두 std namespace 안에 있습니다. 따라서 아래와 같이 3 가지 방법으로 사용할 수 있습니다.

1. 완전한 이름

```
#include <iostream>

int main()
{
    std::cout << std::endl;
}
```

2. using declaration

```
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    std << endl;
}
```

3. using directive

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

int main()
{
    std << endl;
}
```

2 장에서 배운 것 처럼 3 번 처럼 std 이름 공간 전체를 열어 놓을 경우 이름 충돌이 발생할수 있으므로 1 번이나 2 번 방식을 사용하는 것이 좋습니다.

하지만, 이 책에서는 가독성과 지면 관계상 3 번째 방법(using namespace std) 을 사용하도록 하겠습니다.

2.3 C++ 표준 입출력 활용

cout 을 사용해서 변수의 값을 출력할 때 hex, dec, oct 등을 사용하면 16 진수, 10 진수, 8 진수 등으로 출력할 수 있습니다.

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

int main()
{
    int n = 10;
    cout << hex << n << endl;   // 16진수
    cout << dec << n << endl;   // 10진수
    cout << oct << n << endl;   // 8진수
    return 0;
}
```

주의 할 점은 hex, dec, oct 등을 한번 사용하고 나면 변경하기 전까지는 계속 상태가 유지 됩니다.

```
int main()
{
    int n = 10;
    cout << hex << n << endl;   // 'a' 16진수
    cout << n << endl;          // 'a' 계속 16진수로 출력 됩니다.
    cout << dec;                // 10진수로 변경합니다.
    cout << n << endl;          // 10. 이제 10진수로 출력됩니다.
    return 0;
}
```

std::hex, std::dec 등을 흔히 "입출력 조정자(IO manipulator)"라고 부릅니다. 대부분은 <iostream> 헤더를 포함하면 사용할 수 있는데 일부 요소는 <iomanip> 헤더를 포함해야 사용할 수 있습니다.

C++ 표준에는 다음과 같은 "입출력 조정자(IO manipulator)"를 제공하고 있습니다.

<iostream>	boolalpha, noboolalpha, showbase, noshobase, showpoint, noshowpoint, showpos, noshowpos, skipws, noskipws, uppercase, nouppercase, unitbuf, nounitbuf, internal, left, right, dec, hex, oct fixed, scientific, hexfloat, defaultfloat
<iomanip>	setbase, setfill, setprecision, setw, resetiosflags, setiosflags, get_money, put_money, get_time, put_time, quoted

[참고] hex, dec 등은 정확한 정체는 함수입니다. 그런데, hex 가 함수라는 것은 좀 어려운 이야기 이므로 이 책의 후반부(연산자 재정의 항목)에서 자세히 설명하도록 하겠습니다. 지금은 간단한 사용법 정도만 알아 두면 충분합니다.

다음 코드는 입출력 조정자를 사용한 간단한 예제 입니다.

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

int main()
{
    int n = 255;

    cout << hex;
    cout << n << endl;                              / ff
    cout << uppercase << n << endl;                 // FF
    cout << setw(4) << n << endl;                   //   FF
    cout << setw(4) << setfill('0') << n << endl;   // 00FF
```

```
    cout << "##" << setw(6) << "##" << endl;        // ##0000##
    cout << setfill(' ');
    cout << "##" << setw(6) << "##" << endl;        // ##    ##
}
```

3 장. C++ 변수의 특징

3.1 2 진수와 자릿수 표기 - C++11

C++11 부터는 2 진수 표기법도 가능하고, 커다란 숫자를 표기할 때 가독성을 위해서 자릿수를 표기할수도 있습니다.

```
void foo()
{
    int n1 = 0b10;
    int n2 = 1'000'000;
    int n3 = 0b1010'0000;
}
```

현실에서는 자릿수를 콤마(,)연산자를 사용하지만, C 언어에서 이미 콤마 연산자를 사용하고 있기 때문에 C++에서는 싱글 따옴표(')를 사용합니다.

자리수 표기시에 반드시 3 자리마다 해야 하는 규칙은 없습니다. 사용자가 편리한대로 지정하면 됩니다. 보통 현실에서 사용하는 10 진수는 3 자리, 2 진수는 4 자리로 표기하기법을 사용하는 것이 일반적입니다.

3.2 변수 선언의 위치

초창기 C 언어(C89)에서는 변수를 선언할 때 함수의 앞부분 에서만 가능했습니다. 하지만, C++에서는 함수안에서 아무 곳에서나 변수를 선언할 수 있습니다.

```
void foo()
{
    int n1 = 0;
    n1 = 10;
    double d = 3.4; // 오래된 C언어 문법에서는 에러 입니다.
                    // C++ 에서는 에러가 아닙니다.
}
```

물론, 요즘에는 대부분의 C 컴파일러에서도 함수의 중간 부분 에서 변수를 선언할 수 있습니다.

3.3 구조체

C 에서는 구조체 형태의 변수를 만들 때는 구조체 이름 앞에 “struct” 를 붙여야 합니다. 그래서, 흔히 typedef 를 사용하는 경우가 많이 있었습니다. 하지만, C++에서는 구조체를 사용할 때 이름 앞에 struct 를 붙이지 않아도 됩니다.

```
struct Point
{
    int x, y;
};
int main()
{
    struct Point pt1;   // C언어에서는 구조체 이름 앞에 struct 를 붙여야 합니다.
    Point pt2;          // C++에서는 "struct" 붙이지 않아도 됩니다.
}
```

또한, C++11 부터 는 구조체를 만들 때 각 멤버의 초기값을 지정하는 것도 가능합니다.

```
struct Point
{
    int x = 0; // C++11 부터는 구조체를 만들 때 선언에서 초기값 지정이 가능합니다.
    int y = 0; // C언어에서는 할수 없었고, C++09/03 문법에서는 지원하지 못합니다.
};
```

3.4 일관된 변수 초기화 (Uniform initialization) - C++11

C 언어에서는 변수를 초기화 할 때는 "=" 또는 "( )"를 사용합니다. 또한, 배열이나 구조체를 사용할 때는 중괄호("{ }") 를 사용 합니다. 즉, 변수의 종류와 형태에 따라 초기화 방법이 달라집니다.

```
// 일반 변수는 "=" 또는 "( )" 로 초기화 합니다.
int n1 = 0;
int n2(0);

// 배열과 구조체는  "{ }" 로 초기화 합니다.
int x[3] = { 1,2,3 };
struct Point p = { 1,2 };
```

[참고] 변수를 초기화 할 때 C 언어에서는 () 초기화가 안되지만, C++ 부터는 () 초기화가 가능합니다. ()로 초기화하는 것을 "직접 초기화(direct initialization)" 라고 하고, = 로 초기화 하는 것을 "복사 초기화(copy initialization)" 라고 합니다. int 타입과 같은 표준 타입 변수에 대해서는 두 방식의 초기화는 완전히 동일합니다. 하지만, 뒤에 배우는 클래스를 사용한 사용자 정의 타입의 경우 두가지 방식은 차이점이 있습니다. 기본 과정에선 다루기에는 좀 어려운 내용이므로 "C++ Intermediate" 과정을 참고 하시기 바랍니다.

C 언어를 오랫동안 사용해오던 개발자에게는 당연한 개념이지만, C 언어를 처음 배우는 사람에게는 초기화 방식이 다르기 때문에 어려워 보일수 있습니다.

그래서 C++11 부터는 모든 형태의 변수를 초기화 할 때 한가지 방법으로 초기화 할 수 있습니다. **"일관된 초기화(Uniform Initialization)"** 이라고 부르는 문법인데, 중괄호("{ }") 를 사용해서 초기화 합니다.

```
// 모든 종류의 변수를 동일한 방식으로 초기화 합니다.
// 등호(=)를 사용해서 초기화 하고 있습니다. Copy initialization
int   n1    = { 0 };
int   x1[3] = { 1,2,3 };
Point p1    = { 1,2 };

// 등호(=)가 없어도 됩니다. Direct initialization
int   n2 { 0 };
int   x2[3] { 1,2,3 };
Point p2 { 1,2 };
```

[참고] 이 문법의 의도는 C/C++을 오랜 동안 사용한 개발자는 =, (), {} 등의 초기화 방법이 당연하다고 생각하겠지만 C/C++을 처음 배우는 사람은 다양한 스타일의 초기화 방법이 혼란스럽게 생각된다는 점입니다. 그래서, 모든 종류의 변수를 동일한 방법으로 초기화 하자는 의도 입니다.

3.5 preventing narrow - C++11

C++은 C 언어와 마찬가지로 "실수가 정수로의 암시적 형변환"이 가능합니다. 하지만, 이경우는 오히려 data 의 손실이 발생하므로 버그가 발생할 가능성이 많습니다.

"일관된 초기화(Uniform Initialization)"을 사용해서 초기화 하면 data 손실이 발생하는 초기화는 모두 에러가 나오게 됩니다.

```
int n1 = 3.4;       // ok. 하지만 3으로 축소 됩니다. 버그의 원인이 됩니다.
int n2 = { 3.4 };   // error. data손실이 발생되는 초기화는 될 수 없습니다.

char c1 = 300;      // ok. 하지만 overflow 가 발생합니다.
char c2 = { 300 };  // error. 300은 1바이트에 담을 수 없습니다
```

300 의 경우도 1 바이트에 담을 수 없기 때문에 에러가 나오게 됩니다. 이처럼, 변수를 초기화 할 때 "Uniform Initialization" 문법으로 초기화 하면 보다 안전한 코드를 작성할 수 있게 됩니다.

3.6 auto 와 decltype - C++11

auto

C++11 에서 새로 추가된 auto 를 사용하면 우변 수식의 결과값과 동일한 타입의 변수를 만들 수 있습니다.

```
int main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
    auto a1 = x[0]; // 우변인 x[0] 과 동일한 타입(int)의 변수를 a1을 생성합니다.
    auto p = &a1;   // p는 int* 타입으로 결정됩니다.
    auto a2;        // error. 반드시 우변이 있어야 타입을 결정할 수 있습니다.
}
```

[참고] 오래된 C 언어 문법에서 auto 는 정적변수(static)의 반대의 의미로 사용 되었습니다. 이 경우 "auto int a = 0" 방식으로 data type 앞에 auto 를 표기 합니다.

auto 사용의 장점은 배열 x 의 타입을 double 로 변경할 경우 a1, p 의 타입은 자동으로 double, double* 로 결정됩니다. 따라서, 코드의 한 지점을 변경할 때 해당 변수를 참조하는 다른 코드가 자동으로 변경되도록 코드를 작성할 수 있습니다.

또한, auto 는 컴파일시에 우변 수식을 보고 타입을 결정합니다. 결국, 컴파일을 마친 코드는 이미 타입이 결정된 상태이므로 실행시간에 어떠한 성능상의 손해도 없습니다.

decltype

또한, decltype 키워드를 사용하면 기존의 변수와 동일한 타입의 변수를 만들 수 있습니다.

```
int main()
{
    int n1 = 0;
    decltype(n1) n2 = 0;    // n1과 동일한 타입의 변수 n2를 만듭니다.
    decltype(n1) n3;        // 초기값은 없어도 상관없습니다.
}
```

auto 는 우변을 보고 타입을 결정하기 때문에 반드시 초기값이 필요 하지만 decltype 은 괄호안의 변수와 동일한 타입의 변수를 생성하므로 초기값이 없어도 됩니다.

[참고] 사실, auto 와 decltype 을 쉽게 생각할 수 있지만, 생각 보다 어려운 내용이 많이 있습니다. 아래 코드를 컴파일 해보면 에러가 발생합니다. 에러의 원인은 기본 과정에서 다루기는 어려운 내용이므로 "C++ Intemediate"과정을 참고 하시기 바랍니다.

```
int main()
{
    int x[3] = { 1,2,3 };
    auto       p1 = x; // ok.. 아무 문제 없습니다
    decltype(x) p2 = x; // error. 왜 에러 일까요 ?
}
```

3.7 using - C++11

C 에서는 typedef 를 사용하면 기존 타입의 별칭을 만들어서 사용할 수 있습니다.

```
typedef int DWORD;
typedef void(*PF)();

int main()
{
    DWORD n;    // int 타입입니다.
    PF f;       // 함수 포인터 타입입니다.
}
```

C++11 에서 추가된 using 을 사용해도 typedef 처럼 기존 타입의 별칭을 만들수 있습니다.

```
using DWORD = int;
using PF = void(*)();

int main()
{
    DWORD n;    // int 타입입니다.
    PF f;       // 함수 포인터 타입입니다.
}
```

그렇다면 왜 typedef 대신 using 을 사용할까요 ? typedef 은 타입의 별칭만 만들 수 있지만 using 은 타입 뿐 아니라 template 의 별칭도 만들 수 있습니다. 템플릿에 대해서는 아직 자세히 배우지 않았으므로 일단은 typedef 대신으로 using 을 사용할 수 있다는 정도만 알아 두면 될 거 같습니다.

3.8 64 비트 타입 - C++11

C++11 부터는 64 비트 정수 타입인 long long 타입을 지원합니다.

```
long long a = { 3 };
cout << sizeof(a) << endl; // 64
```

3.9 constexpr - C++11

변수 선언시 const 키워드를 사용하면 값을 변경할수 없는 변수를 만들수 있습니다.

```
int main()
{
    const int c = 10;   // c는 상수 입니다.
    c = 20;             // error
}
```

C++11 에서 추가된 constexpr 을 사용해도 동일한 효과를 볼수 있습니다.

```
int main()
{
    constexpr int c = 10;   // C++11 constexpr을 사용해도 상수를 만들수있습니다.
    c = 20;                 // error
}
```

두 키워드의 차이점은 const 는 컴파일 시간 상수 뿐 아니라 실행시간에 변경된 변수로도 초기화 할수 있지만 constexpr 은 컴파일 시간 상수만 가능하다는 점입니다.

```
int main()
{
    // 아래 코드는 컴파일 시간에 컴파일러가 10 이라는 값을 알게 됩니다.
    const     int c1 = 10; // ok
    constexpr int c2 = 10; // ok

    int n = 0;
```

```
    cin >> n;       // n의 값은 실행해서 사용자에게 입력 받아야만 얼마인지 결정됩니다.

    const     int c3 = n;   // ok. c3는 실행시 결정된 값으로 초기화 될수 있습니다.
    constexpr int c4 = n;   // error. constexpr은 컴파일 시간에 알수 있는값
                            //(컴파일 시간 상수)으로만 초기화 가능합니다.
}
```

[참고] "컴파일 시간 상수" 가 왜 중요 할까? 라는 궁금증을 가진 독자도 계실거 같습니다. C++의 어려운 분야중 하나인 "template" 분야에서는 컴파일 시간에 연산을 하게 하는 메타 프로그래밍 분야가 있습니다. 이 분야에서는 실행시간 상수가 아닌 컴파일 시간 상수가 아주 중요하게 됩니다. 더 자세한 이야기는 "C++ Template" 과정을 참고 하시면 됩니다. 또한, 컴파일러 최적화 관점에서도 많은 차이점이 있습니다.

3.10 structured binding declaration - C++17

C++17 에서는 구조체나 배열등 에서 값을 꺼낼 때 사용하는 "structured binding declaration" 문법을 제공합니다.

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

struct Point
{
    int    x{ 10 }; // C++11 부터 사용가능합니다.
    double y{ 3.4 };
};
int main()
{
    Point p;
    int x[2] = { 1,2};

    // 구조체 p에 있는 값을 a, b 변수로 꺼내 옵니다.
    auto[a, b] = p; // auto a = p.x;
                    // auto b = p.y;
    // 배열에서도 꺼낼수 있습니다.
    auto[c, d] = x; // auto c = x[0];
                    // auto d = x[1];

    cout << a << ", " << b << endl;
    cout << c << ", " << d << endl;
};
```

[참고] C++17 문법이므로 g++ 사용시 "-std=c++1z" 옵션을, VC++(cl.exe) 사용시 "/std:c++latest" 옵션을 지정해야 합니다.

"structured binding declaration" 문법은 다음과 같은 형식으로 사용합니다.

"auto [변수, 변수,...] = 배열 또는 구조체(클래스)"

"suctured binding declaration" 문법을 사용 할 때는 변수의 타입은 반드시 auto 를 사용해야 합니다. auto 이외의 타입을 사용하면 에러가 발생합니다. 또한, 구조체가 아닌 공용체(union)에는 사용할수 없습니다.

```
int main()
{
    int x[2] = { 1,2};

    int[c, d] = x;  // error. int 대신 auto를 사용해야 합니다.

    cout << c << ", " << d << endl;
}
```

3.11 문자열

C 에서는 문자열을 처리하기 위해 char* 또는 char 배열을 사용 하지만 C++에서는 string 타입을 사용하면 됩니다. string 타입을 사용하려면 <string> 헤더가 필요합니다.

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

int main()
{
    // C언어에서는 char 타입의 배열이나 포인터를 사용해서 문자열을 처리합니다.
    char s1[] = "hello";
    const char* s2 = "hello";

    // C++에서는 string 타입으로 문자열을 처리합니다.
    string s3 = "hello";
}
```

[참고] string 은 정확히는 기본 타입이 아닌 class 로 만든 사용자 정의 타입 입니다. C++의 표준 라이브러리인 STL 에 포함된 라이브러리 입니다. 자세한 이야기는 "2 장 객체지향 프로그래밍"에서 자세히 다루겠습니다.

C 에서 문자열을 다루려면 strcpy(), strcmp()등 문자열 함수를 사용해야 했는데, string 타입을 사용하면 문자열 조작을 직관적으로 편리하게 다룰 수 있습니다.

```
int main()
{
    string s1 = "hello";
    string s2 = s1; // strcpy 함수를 사용할 필요가 없습니다.

    if ( s1 == s2 ) // strcmp 함수를 사용할 필요가 없습니다.
    {
    }
}
```

비록, string 타입이 아직 배우지 않은 class 문법으로 만든 사용자 정의 타입이지만 정확한 원리를 몰라도 사용하는 것은 어렵지 않으므로 예제를 만들 때 사용하도록 하겠습니다. 정확한 원리에 대해서는 "2 장. 객체 지향 프로그래밍"에서 다루 도록 하겠습니다.

4장. C 와는 다른 C++ 함수

함수는 C 언어에서 가장 중요한 요소 중 하나 입니다. C++에서는 C 함수의 모든 문법을 지원하고 추가로 다양한 문법을 제공하고 있습니다. 이번 장에서는 C++에서 추가된 함수 관련 다양한 문법들을 살펴 보도록 하겠습니다.

4.1 디폴트 파라미터 (default parameter)

C++에서는 함수를 호출할 때 인자를 전달하지 않으면 디폴트 값을 사용하도록 할 수 있습니다.

```
void foo(int a, int b = 2, int c = 3)
{
    cout << a << " " << b << " " << c << endl;
}
int main()
{
    foo(1, 1, 1);   // 1, 1, 1
    foo(1, 1);      // 1, 1, 3
    foo(1);         // 1, 2, 3
    foo();          // error
}
```

디폴트 파라미터를 사용할 때는 다음의 2 가지를 주의 해야 합니다.

주의 1. 반드시 마지막 파라미터부터 차례대로 디폴트 값을 적용해야 합니다.

```
void foo(int a,     int b = 2, int c = 3);  // ok
void foo(int a = 1, int b = 2, int c = 3);  // ok
void foo(int a = 1, int b,     int c = 3);  // error
void foo(int a = 1, int b = 2, int c);      // error
```

주의 2. 함수를 선언과 구현으로 분리 할 때는 선언부에만 디폴트 값을 표기해야 합니다.

```
void foo(int a, int b = 2, int c = 3);  // 선언부에 디폴트 값을 표기 합니다.

int main()
```

```
{
    foo(1);
}
void foo(int a, int b = 2, int c = 3)   // error. 구현부에도 디폴트 값을 표기하면
{                                       // 컴파일 에러 입니다
    cout << a << " " << b << " " << c << endl;
}
```

참고로, 함수 선언부에 디폴트 값을 표기하지 않고 구현부에만 표기하면 함수 자체는 에러가 아니지만 디폴트 값을 사용할 수 없게 됩니다. 컴파일러는 함수 호출시에 함수의 선언 모양을 보고 문법적 오류를 확인 하게 되는데, 선언부에 디폴트 인자가 없으므로 호출시에는 반드시 인자를 전부 보내야 하기 때문입니다.

```
void foo(int a, int b, int c);  // 선언부에 디폴트 값이 없습니다.

int main()
{
    foo(1, 2, 3);   // ok. 모든 인자를 다 전달하면 문제 없습니다.
    foo(1);         // error. 컴파일러가 이부분을 컴파일 할 때 함수 선언부를
                    // 통해서 디폴트 값이 없는 것으로 알고 있습니다.
}
void foo(int a, int b = 2, int c = 3)
{
    cout << a << " " << b << " " << c << endl;
}
```

결국, 디폴트 값은 선언부에만 값을 표기해야 하는데, 구현부에는 주석을 사용해서 디폴트 값이 있음을 알려주는 경우가 많이 있습니다.

```
void foo(int a, int b = 2, int c = 3);
void foo(int a, int b /*= 2*/, int c /*= 3*/ )
{
    cout << a << " " << b << " " << c << endl;
}
```

4.2 함수 오버로딩 (function overloading)

C++에서는 파라미터의 타입이나 개수가 다르면 동일한 이름의 함수를 여러 개 만들 수 있습니다. "함수 오버로딩(function overloading)"이라고 불리는 문법입니다.

```
// 파라미터의 타입이 다르므로 동일한 이름의 함수를 만들 수 있습니다.
int square(int a)
{
    return a * a;
}
double square(double a)
{
    return a * a;
}
int main()
{
    square(3);
    square(3.3);
}
```

위 코드에서 square 함수는 분명 2 개 이지만, square 함수를 사용하는 사용자 입장에서는 마치 동일한 함수 처럼 생각하고 사용할 수 있습니다. 함수 오버로딩(function overloading) 덕분에 일관된 형태의 라이브러리를 구축할 수 있게 되었습니다. C 언어 에서는 동일한 이름을 2 개 만들 수 없기 때문에 함수의 이름을 다르게 만들어야 합니다. 물론, 매크로 함수를 사용하면 함수 오버로딩과 유사한 효과를 낼 수 있지만, 매크로 함수는 문자 치환 방식이기 때문에 버그의 확률이 높습니다.

```
// C언어 에서는 동일한 이름의 함수를 여러 개  만들 수 없으므로
// 함수의 이름을 변경해야 합니다.
int    square_int(int a)       { return a * a; }
double square_double(double a) { return a * a; }

// 매크로 함수로 유사한 효과를 낼 수 는 있습니다.
#define SQUARE(x)  (x) * (x)

int main()
{
    int n = 3;
    printf("%d\n", SQUARE(++n));    // 결과는 얼마 일까요 ?
                                    // 매크로 함수는 버그가 나올 확률이 높습니다.
}
```

함수 오버로딩시 주의할 점은 파라미터의 타입이나 개수가 다른 경우는 동일 이름의 함수를 만들 수 있지만 리턴 타입만 다른 경우는 만들 수 없습니다. 또한, 디폴트 파라미터가 있는 경우도 주의 해야 합니다.

```
// 아래 3개의 함수는 아무 문제 없습니다.
void foo(int a)         {}
void foo(double d)      {}
void foo(int a, int b)  {}

// 잘못된 코드 입니다. 인자의 갯수가 다르지만 디폴트 파라미터가 있습니다.
void goo(int a)             {}
void goo(int a, int b = 0)  {}

goo(3); // 이순간 어떤 함수를 호출 할지 컴파일러가 알 수 없습니다.

// 역시 잘못된 코드 입니다.
void hoo(int a) {}
int  hoo(int a) {}

hoo(3); // 어떤 함수와 연결할 것인가를 결정할 때는 리턴 타입을 영향을 주지 않습니다.
```

4.3 인라인 함수 (inline function)

일반적으로 함수를 호출하면 함수 인자를 스택에 넣고 해당하는 함수로 이동(jmp) 합니다. 그리고, 함수의 코드를 수행한 후에 다시 호출한 곳으로 되돌아 오게 됩니다. 하지만, C++ 의 인라인 함수를 사용하면 함수 호출 시 실제로 함수 부분으로 이동하지 않고, 함수의 기계어 코드 전체를 치환 할 수 있습니다.

```
int Add1(int a, int b)
{
    return a + b;
}
inline int Add2(int a, int b)
{
    return a + b;
}
int main()
{
    int n1 = Add1(1, 2); // 이순간 Add1 함수로 이동(jmp) 합니다.
    int n2 = Add2(1, 2); // 이순간 Add2 의 기계어 코드를 이 부분에 치환합니다.
}
```

인라인 함수의 **장점과 실제 함수로 이동하는 것이 아니므로 속도가 빨라** 집니다. 하지만, 단점은 함수 몸체가 큰 경우 여러 번 호출하면 결국 여러 곳에 함수 코드가 치환 되므로 목적코드(실행파일)의 크기가 커질 수 있습니다. 따라서, 인라인 함수는 크기가 작은 1~2 줄 짜리의 함수에 주로 사용합니다.

4.4 템플릿 (template)

동일한 이름의 함수를 여러 개 만들 수 있는 함수 오버로딩 덕분에 사용하기 쉬운 일관성 있는 라이브러리 만들 수 있습니다.

```
// 리턴 타입과 파라미터 타입을 제외하면 구현이 완전히 동일합니다.
int    square(int a)   { return a * a;}
double square(double a){ return a * a;}

int main()
{
    // 사용자 입장에서는 같은 함수 처럼 보입니다.
    square(3);
    square(3.3);
}
```

하지만, 위 코드는 사용자 입장에서는 square 함수가 하나 처 럼 보이지만 square 를 만드는 사람 입장에서는 결국 2 개를 만들어야 합니다. 그런데, 이 2 개의 함수는 인자의 타입과 리턴 타입을 제외하면 구현이 완전히 동일 합니다. 이처럼 유사한 코드가 반복될때는 템플릿을 사용하면 편리 합니다.

템플릿은 함수를 만드는 틀(Template)을 제공하는 것입니다. 사용자가 틀을 제공하면 컴파일러가 소스 코드를 컴파일 할 때 어떤 타입의 함수가 필요한지를 조사해서 실제 함수를 생성하는 것입니다.

템플릿을 만들 때는 아래 모양으로 하게 됩니다.

```
template<typename T>    // 함수가 아닌 함수를 만드는 틀(template)임을 컴파일러에게
T square(T a)           // 알려 줍니다. 타입이 필요한 자리에 T를 적습니다.
{
    return a * a;
}
```

이제, 사용자가 square 를 사용하면 컴파일러가 인자의 타입을 결정해서 해당하는 함수를 생성하게 됩니다.

```
int main()
{
    square(3);  // 3은 int 이므로 컴파일러가 square 템플릿을
```

```
                // 사용해서 T를 int로 변경한 함수를 생성합니다.

    square(3.3);// 3.3은 double 이므로 컴파일러는
                // double square(double) 함수를 생성합니다.
}
```

결국, 최종적으로 컴파일 된 코드 안에는 square 함수는 2 개가 있게 됩니다.

✍ **사용자가 작성한 코드**

```
template<typename T>
T square(T a)
{
    return a * a;
}

int main()
{
    square(3);
    square(3.3);
}
```

✍ **컴파일러에 의해 생성된 코드**

```
int square(int a)
{
    return a * a;
}
double square(double a)
{
    return a * a;
}
int main()
{
    square(3);
    square(3.3);
}
```

템플릿(template)을 만들 때 T 대신 다른 문자를 사용해도 전혀 문제 없습니다. 관례적으로 T 를 많이 사용할 뿐입니다. 또한, typename 대신 class 키워드를 사용해도 동일합니다.

```
template<typename  TYPE  >  // T 대신 아무 문자나 사용해도 됩니다. 여러 글자도 됩니다.
TYPE square(TYPE a)
{
    return a * a;
}

template<class  TYPE  > // typename 대신 class를 사용해도 됩니다.
TYPE add(TYPE a, TYPE b)
{
    return a + a;
}
```

4.5 함수 삭제 (delete function) - C++11

C 에서는 double 타입의 값도 int 타입으로 암시적 형변환이 됩니다. 그런데, 이런 특징 때문에 오히려 버그가 나올 확률이 높아 집니다.

```
// 두개 정수의 최대 공약수를 구하는 함수
int gcd(int a, int b)
{
    return b != 0 ? gcd(b, a % b) : a;
}
int main()
{
    gcd(2, 10);
    gcd(2.2, 4.3);  // 2.2는 double이지만 int로 암시적 형변환이 가능합니다.
                    // 하지만 결국 버그입니다.
}
```

물론, 일부 컴파일러는 "데이터 손실이 발생 할 수 있다" 는 경고를 내는 경우도 있습니다. 이때, 이와 같은 암시적 변환에 따른 호출을 막으려면 몇가지 방법이 있습니다.

- **의도적으로 함수의 선언부만 제공하는 방법.**

gcd(double, double) 버전의 함수를 의도적으로 선언부만 제공 하므 로서 gcd(2.2, 4.3)이 link 에러가 나오게 할 수 있습니다.

```
// 두개 정수의 최대 공약수를 구하는 함수
int gcd(int a, int b)
{
    return b != 0 ? gcd(b, a % b) : a;
}

double gcd(double a, double b); // 의도적으로 함수의 선언만 제공합니다.

int main()
{
    gcd(2, 10);
    gcd(2.2, 4.3);  // gcd(double, double) 버전의 선언이 있으므로 컴파일시에는
                    // 문제가 없습니다. 하지만, gcd(double, double)의  구현이
                    // 없으므로 link 시 에러가 발생합니다.
                    // gcd(double, double) 을 찾을 수 없다는 link error 입니다.
```

```
}
```

- **함수를 삭제하는 방법.**

또, 다른 방법은 C++11 부터 등장한 "delete function" 이라는 문법을 사용해서 함수를 삭제하는 방법입니다.

```
int gcd(int a, int b)
{
    return b != 0 ? gcd(b, a % b) : a;
}
double gcd(double a, double b) = delete; // 함수를 삭제 합니다.

int main()
{
    gcd(2, 10);
    gcd(2.2, 4.3);  // delete된 함수를 사용하므로 컴파일 시간 에러 입니다.
}
```

"delete function" 문법은 함수 선언문 뒤에 "= delete" 를 표시하면 됩니다.

선언만 제공하는 것과 삭제 하는 것의 차이점은, 선언만 제공시 컴파일시에는 에러가 없고 링킹할 때 에러가 발생합니다. 하지만, "delete function"을 사용하면 컴파일 시에 에러가 나오게 할 수 있습니다. 완전한 실행 파일이 아닌 라이브러리(.dll, .lib, .a, .so)등을 만들 때 좀더 안전하게 사용할 수 있습니다.

중요한 것은 특정함수를 제공하지 않은 것, 선언만 제공한 것, 삭제 한 것에 대한 차이점을 명확이 알아야 합니다.

double 버전을 제공하지 않을 때	**암시적 변환에 의해서 호출가능한 함수가 사용됩니다. double 은 int 로 변환 될 수 있으므로 int 버전의 함수가 사용됩니다.**
double 버전을 선언만 제공할 때	**double 버전을 사용하면 링크 에러가 발생합니다.**
double 버전을 삭제(delete)할 때	**double 버전을 사용하면 컴파일 에러가 발생합니다.**

4.6 후위 반환 형식 (trailing return, suffix return type)

일반적으로 함수를 만들 때는 리턴 타입을 함수 이름 앞에 적게 됩니다.

```
// 전통적인 모양의 함수.
// "리턴타입  함수이름(파라미터 리스트)" 의 모양으로 만들게 됩니다.
int square(int a)
{
    return a * a;
}
```

하지만, C++11 부터는 리턴 타입을 함수의 파라미터 리스트 뒤에 적는 새로운 문법이 지원 됩니다. "후위형 리턴 타입(trailing return type)" 이라고 불리는 문법인데, 함수 이름 앞에는 auto 를 적고 함수 파라미터 리스트 뒤에 ->를 표기하고 리턴 타입을 적게 됩니다. 정확한 모양은 아래와 같습니다.

```
// 새로운 함수 모양
// "auto 함수이름(파라미터리스트) -> 리턴 타입"
auto square(int a) -> int
{
    return a * a;
}
```

기존의 함수 모양이 어떤 문제점이 있어서 이런 문법이 나왔을까요 ?

아직은 배우지 않았지만 C++의 복잡한 문법 중 하나인 템플릿 이나 람다 등을 사용할 때 기존의 함수 모양으로는 해결할 수 없는 문제 점이 있었기 때문에 새로운 모양의 함수가 나오게 되었습니다. 참고로 애플이 만든 새로운 언어인 "swift" 에서도 함수를 만들 때 후위형 반환 타입의 모양을 사용합니다.

5장. 새로운 반복문과 제어문

5.1 ranged for - C++11

C++11 부터 전통적인 for 문 외에 ranged-for 문을 추가로 제공합니다.

```
int main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };

    // 전통적인 for 문
    for (int i = 0; i < 10; i++)
        cout << x[i] << endl;

    // 새로운 ranged for 문
    for (int n : x)     // 배열 x 에서 요소를 하나씩 꺼내서 n에 담아 줍니다.
        cout << n << endl;
}
```

java, C#등의 foreach 와 유사한 구문인데, 정확한 사용법은 다음과 같습니다.

```
for (DataType 변수 : 배열 또는 STL 컨테이너 )
{
}
```

[참고] ranged - for 문의 내부적인 원리는 다소 복잡하므로 이 책에서는 다루지 않습니다. "C++ Intermediate"과정을 참고 하시기 바랍니다.

배열 뿐 아니라 이 책의 후반부에서 배우는 STL 의 다양한 컨테이너도 사용 가능 합니다. 또한, auto 를 사용하면 보다 편리하게 사용할 수 있습니다.

```
for (auto n : x)
    cout << n << endl;
```

5.2 init-control statement - C++17

C 프로그래밍에서 함수 호출뒤 결과값을 if 문으로 조사하는 코드는 아주 널리 사용됩니다.

```
// 함수 결과값을 조사하는 전통적인 방식의 코드
int ret = foo();

if (ret == 0)
{
}
```

C++17 에 추가된 "초기화 구문이 추가된 제어문(if-statement with initializer)" 을 사용하면 보다 간결하게 표현할수 있습니다.

```
int foo()
{
    return 0;
}

int main()
{
    // 전통적인 방식의 함수 결과값 조사
    int ret = foo();

    if (ret == 0)
    {
    }

    // C++17의 if-init을 사용한 코드
    if (int ret2 = foo(); ret2 == 0)
    {
        // ret2는 블럭 안에서만 유효 합니다.

    } // 이순간 ret2는 파괴 됩니다.
}
```

새로운 if 문의 정확한 사용법은 다음과 같습니다.

If (초기화 구문 ; 조사식)

또한, 초기화 구문 안에서 선언된 변수는 if 블록 안에서만 유효하며 블록을 벗어나면 파괴됩니다.

```
int foo()
{
    return 0;
}
int main()
{
    // C++17의 if-init을 사용한 코드
    if (int ret2 = foo(); ret2 == 0)
    {
        // ret2는 블럭 안에서만 유효 합니다.

    } // 이순간 ret2는 파괴 됩니다.
}
```

if 문 뿐 아니라 switch~case 문도 초기화 구문을 지정할 수 있습니다.

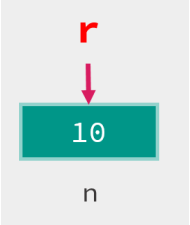
```
int foo()
{
    return 0;
}
int main()
{
    // C++17의 if-init을 사용한 코드
    if (int ret2 = foo(); ret2 == 0)
    {
        // ret2는 블럭 안에서만 유효 합니다.

    } // 이순간 ret2는 파괴 됩니다.
}
```

6장. Reference

6.1 reference 개념

C++에는 레퍼런스(reference) 타입이라는 새로운 형태의 타입을 제공합니다. 레퍼런스 타입은 새로운 메모리를 할당하는 것이 아니고 기존 메모리에 새로운 이름을 부여 할 때 사용합니다.

```
int main()
{
    int n = 0;  // 이순간 4바이트의 메모리가 할당 됩니다.

    int& r = n; // 새로운 메모리를 할당하는 것이 아니라
                // 기존 메모리(n)의 새로운 이름 r을
                // 부여합니다.
    r = 10;     // r와 n은 같은 메모리 이므로
                // 결국 n에 10이 들어갑니다.

    cout << n << endl; // 10
}
```



레퍼런스 변수를 만드는 방법은 다음과 같습니다.

```
int& r = n; // DataType& 레퍼런스변수명 = 기존변수;
```

이 경우, 중요한 것은 새로운 메모리가 만들어 지는 것이 아니라 기존 변수와 동일 메모리를 사용하게 된다는 점입니다. 즉, 기존 변수에 대한 별칭(alias)를 만드는 과정입니다.

따라서, r 의 값을 변경하면 자동으로 n 의 값도 변경됩니다. 두 변수의 주소값을 출력해도 동일한 주소가 나오는 것을 볼 수 있습니다.

```
int main()
{
    int  n = 0;
    int& r = n;

    cout << &n << endl;
    cout << &r << endl;
}
```

참고로, 변수를 사용할 때 이름 앞에 &를 붙이면 주소연산자로서 메모리 주소를 꺼내는 표현이지만, 변수 선언시 &를 사용하는 것은 레퍼런스 변수를 만드는 표현이므로 잘 구별해 놓으시기 바랍니다.

```
int main()
{
    int  n = 0;
    cout << &n << endl; // 변수 이름 앞에 & 는 주소연산자 입니다.

    int& r = n; // 레퍼런스 변수를 선언하기 위해 필요한 & 입니다.
}
```

레퍼런스 변수는 기존 변수의 별칭(alias) 이므로 반드시 초기값이 있어야 합니다.

```
int main()
{
    int  n = 0;
    int& r1 = n;// ok, r1은 n의 별칭입니다.
    int& r2;    // error. 초기값이 없습니다.
}
```

6.2 call by value, call by pointer, call by reference

C 언어에서 함수에 인자를 전달 할 때 값으로 보내면 복사본이 만들어 지므로 인자 값을 변경할 수는 없게 됩니다. 함수에 보낸 인자의 값을 변경 할 수 있게 하려면 값이 아닌 주소를 보내야 합니다.

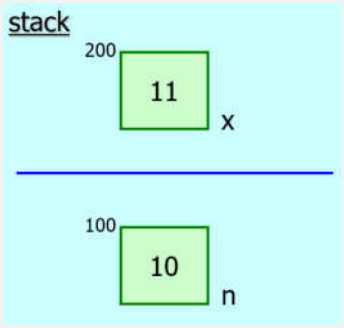
흔히, call by value, call by pointer 로 불리는 개념인데, C++에서는 reference 라는 타입이 별도로 있으므로 call by pointer 라고 하겠습니다.

함수 인자를 레퍼런스로 보내면 함수에서 값을 변경 시 원본도 변경되게 할 수 있습니다.

아래 3 개의 코드의 각각의 메모리 그림을 잘 생각해 봅시다.

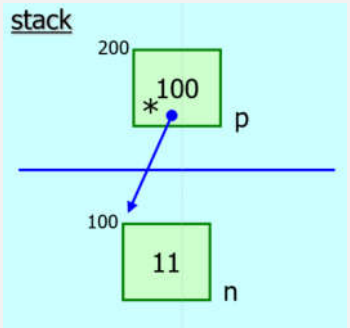
1. call by value

```
void inc(int x)
{
    ++x;
}
int main()
{
    int n = 10;
    inc(n);
    cout << n << endl;
}
```



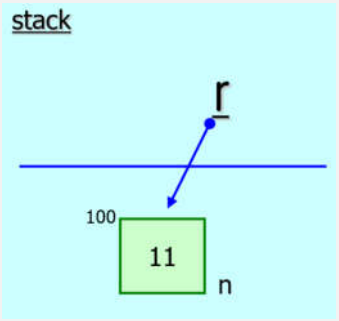
2. call by pointer

```
void inc(int* p)
{
    ++(*p);
}
int main()
{
    int n = 10;
    inc(&n);
    cout << n << endl;
}
```



3. call by reference

```
void inc(int& r)
{
    ++r;
}
int main()
{
    int n = 10;
    inc(n);
    cout << n << endl;
}
```



위 코드에서 1 번의 경우 함수 인자에 대한 복사본을 생성하게 되므로 원본 n 을 변경 할 수 없습니다.

2 번은 주소를 보내서 변경하고 있으므로 원본 n 이 변경됩니다. 마지막으로 3 번은 참조 변수 (reference) 를 사용하고 있으 므로 새로운 메모리가 아닌 기존 메모리의 별칭으로 전달 받게 되므로 원본을 변경할 수 있게 됩니다.

결국, call by pointer 와 call by reference 는 둘다 자로 보낸 변수 값을 변경할 수 있게 됩니다. 그렇다면, call by pointer 와 call by reference 중 어떤 것이 좋을까요 ? 둘다 널리 사용되는 방법이긴 하지만 다음과 같은 의견도 있습니다. 아래 코드를 생각해 봅시다.

```
void inc1(int* p) { ++(*p); }
void inc2(int& r) { ++r; }

int main()
{
    int x = 10, y = 10;
    inc1(&x); // 주소를 보내고 있습니다. 값이 변경될 것 이라고 예측할 수 있습니다.
    inc2(y);  // call by value 인지 call by reference 인지 알 수 가 없습니다.
}
```

즉, 함수 인자를 참조로 전달 받을 경우,함수를 호출하는 코드 만으로는 값이 변경될 것인지를 알 수 없다는 "코드 가독성"의 문제가 있습니다.

scanf 함수는 사용자 입력을 담아 오기 위해 변수의 주소를 전달 해야 합니다. 하지만, cin 은 입력 값을 담아 오기 위해 주소를 전달할 필요가 없습니다. 그 이유는, cin 이 변수를 전달 받을 때 참조로 전달 받고 있기 때문입니다.

```
scanf("%d", &n); // 주소를 전달합니다. call by pointer 입니다.
cin >> n;        // 주소가 아닙니다.  call by reference 입니다.
```

[참고] "cin >> n" 은 "cin.operator>>()" 의 원리 입니다. 자세한 원리는 연산자 재정의를 다룰 때 이야기 하도록 하겠습니다.

6.3 call by value 와 const &

함수 인자를 call by value 로 전달 받으면 복사본을 만들게 되므로 인자로 보낸 변수를 절대 변경할 수 없습니다. 즉, 인자가 변경되지 않는다는 것을 보장 받고 싶으면 함수를 만들 때 call by value 로 만들면 됩니다.

```
void foo(int n)
{
    n = 20; // 복사본이 변경되므로 main 함수의 변수 x는 변경되지 않습니다.
}
int main()
{
    int x = 10;
    foo(x); // 함수 호출 뒤에도 x가 변화가 없다는 것을 보장 받으려면
            // call by value로 만들면 됩니다.
}
```

하지만, call by value 는 함수 인자에 대한 복사본을 만들게 되므로 메모리에 대한 부담이 있습니다. int , double, char 등의 기본 타입은 크기가 크지 않으므로 문제가 되지는 않습니다. 하지만, 구조체 등 비교적 크기가 큰 사용자 정의 타입을 사용하면 문제가 될 수 있습니다. 이때는, 참조 변수(reference)를 사용하는 것이 좋습니다. 하지만, 참조 변수로 전달 받으면 원본 인자가 변경될 수 있다는 문제가 있습니다. 해결책은 const 참조를 사용하는 것 입니다.

```
struct BigData
{
    int data;
    //...
};
void foo(BigData d1) {}     // 복사본 d1을 만들게 되므로 메모리 부담이 있습니다.

void goo(const BigData& d1) // 참조 이므로 메모리 부담이 없습니다
{
    d1.data = 0;            // error. const 참조 이므로 변경 할 수 없습니다.
}
int main()
{
    BigData d;
    foo(d);
}
```

함수 인자로 call by value 대신 const &를 사용하는 것은 C++에서는 너무나 유명한 관례 이므로 반드시 정확히 이해 해야 합니다. 단, int, double, char 등의 사용자 정의 타입이 아닌 primitive 타입은 메모리 사용량이 크기 않고, 컴파일러 최적화 등의 이유로 const 참조가 아닌 call by value 로 전달 받는 것이 좋습니다.

```
void foo(int n) {}              // primitive type은 call by value를 사용합니다.
void foo(const BigData& n) {}   // user define type은 const 참조를 사용합니다.
```

6.4 rvalue reference - C++11

이번 주제는 좀 복잡하고 어렵습니다. 되도록 간결하게 핵심만 정리 하겠습니다. 깊이 있는 내용은 "C++ Intermediate" 과정을 참고 하시기 바랍니다.

C/C++에서 등호(=)의 왼쪽에 올 수 있는 것을 lvalue, 왼쪽에 올 수 없는 것은 rvalue 라고 합니다.

```
int main()
{
    int n1 = 0;
    int n2 = 0;

    n1 = 10;    // ok. n1은 왼쪽에 놓을수 있습니다. lvalue 입니다.
    10 = n1;    // error. 10은 왼쪽에 놓을수 없습니다. rvalue 입니다.

    n2 = n1;    // n1은 오른쪽에 놓을수도 있습니다.
                // lvalue는 왼쪽과 오른쪽에 모두 사용될수 있습니다.
}
```

참조 변수로 lvalue 는 참조 할 수 있지만, rvalue 는 참조 할 수 없습니다. 하지만, const 참조를 사용하면 lvalue, rvalue 를 모두 참조 함수 있습니다.

```
int main()
{
    int n = 0;

    int& r1 = n;  // ok. lvalue를 가르킬수 있습니다.
    int& r2 = 10; // error. rvalue를 가르킬수 없습니다.
```

```
    // 상수 참조는 rvalue, lvalue를 모두 가르킬수 있습니다.
    const int& r3 = n; // ok
    const int& r4 = 10;// ok
}
```

C++11 에서는 기존의 참조 변수 외에 rvalue reference 라는 새로운 개념을 도입했습니다. rvalue reference 는 rvalue 만 참조 할 수 있습니다. rvalue reference 를 만들때는 & 연산자를 2 개 사용합니다.

```
int main()
{
    int n = 0;

    int&& r1 = n;  // error. lvalue를 가르킬수 없습니다.
    int&& r2 = 10; // ok.    rvalue를 가르킬수 있습니다.
}
```

또한, rvalue reference 의 등장으로 기존에 있는 reference 는 "lvalue reference"라고 부르게 됩니다.

C++을 처음 배우시는 분들은 이부분을 읽으시면 왜 rvalue 를 담을 수 없고, 왜 const 참조는 rvalue 를 담을수 있고 rvalue reference 는 또 뭔가? 라는 의문이 들 수 있습니다. 좀 복잡해지는 내용이므로 이번 과정에서는 아래 내용만 정확히 알아 두시고, 이 곳에 담긴 철학은 "Advanced C++" 과정을 참고 하시기 바랍니다. reference 는 생각보다 어려운 내용이 많이 있습니다.

- int& : lvalue reference 입니다. lvalue 만 참조 할 수 있습니다.
- int&&: rvalue reference 입니다. rvalue 만 참조 할 수 있습니다.
- const int& : lvalue, rvalue 를 모두 참조 할 수 있습니다.

7 장. C++ Explicit Casting

7.1 C Casting 의 문제점.

C 언어에는 암시적(implicit) 변환과 명시적(explicit) 변환이라는 개념이 있습니다.

```
int main()
{
    double d = 3.4;
    int n = d;      // ok. d는 double 이지만 int로 암시적 변환 됩니다.

    int* p = &d;    // error. double*를 int*로 암시적 변환되지 않습니다.
    int* p = (int*)&d; // ok. 명시적 변환은 허용됩니다.
}
```

그런데, C 의 명시적 캐스팅은 대부분 성공하기 때문에 버그의 원인이 되는 경우가 많이 있습니다. 아래 코드를 생각해 봅시다.

```
int main()
{
    int n = 0; // 4바이트 크기의 메모리 입니다

    double* p = (double*)&n; // 명시적으로 double* 타입으로 캐스팅 합니다.
                             // 컴파일러는 p가 가르키는 곳을 double로 생각하게 됩니다.
    *p = 3.4;   // bug!!. p가 가르키는 곳에 3.4를 넣게 됩니다.
                // 그런데, p를 double*로 생각하므로 8바이트 크기로 넣게 됩니다.
}
```

위 코드에서 n 은 4 바이트 크기의 메모리 이지만 *p = 3.4 로 넣을 때는 8 바이트 공간으로 넣게 된다는 점입니다. 실행시간에 잘못된 메모리 참조로 죽을 수도 있지만, 더 큰 문제는 다른 변수가 사용하던 공간을 덮어 쓸 수 도 있다는 점입니다. 이 문제의 가장 근본적인 원인은 int 형 변수의 주소를 double 포인터 타입으로 캐스팅을 허용한다는 점입니다.

```
double* p = (double*)&n;    // 이 캐스팅은 상당히 위험합니다.
                            // 하지만 C언어에서는 허용합니다.
```

해결책을 배우기 전에 또 다른 문제점을 살펴 보겠습니다. 아래 코드를 생각해 보세요.

```
int main()
{
    const int c = 10;

    // c = 20;      // error. 상수 이므로 값을 변경 할 수 없습니다.
    // int* p = &c; // error. 상수의 주소를 int*에 담을 수는 없습니다.

    int* p = (int*)&c;  // ok. 하지만 명시적 캐스팅을 사용하면 에러가 나지 않습니다.
                        // int* p 이므로 컴파일러는 p가 가르키는 곳을 int
                        // 즉, 상수가 아닌 것 으로 생각합니다.

    *p = 20;            // p가 가르킨느 곳이 상수가 아닌것으로 생각하므로
                        // 에러가 아닙니다.
    cout << c << endl; // 얼마가 나올까요 ?
}
```

[참고] 위 코드를 실행해 보고 결과를 확인해 보세요. 그리고 왜 c 의 값이 그렇게 나오는지를 생각해 보세요.

위 코드는 상수 c 를 만들고 주소를 명시적 캐스팅을 통해 int* 로 변경한후 값을 변경하는 코드입니다. 변경 할 수 없는 상수를 만들고 강제로 변경하고 있는 것입니다. 결국 문제는 아래 캐스팅을 허용한다는 점입니다.

```
int* p = (int*)&c;  // 상수의 주소를 비상수를 가르키는 포인터에 담는 것은 위험합니다.
                    // 하지만, C언어 에서는 허용합니다.
```

7.2 C++ explicit Casting

C++ 에서는 C 캐스팅의 문제를 해결하기 위해 4 개의 캐스팅 연산자를 제공합니다.

static_cast<>	가장 기본 적인 캐스팅입니다. 컴파일 시간에 캐스팅을 수행하기 때문에 static_cast<>라고 부릅니다. 서로 연관성이 없는 타입의 캐스팅은 허용되지 않습니다.
dynamic_cast<>	실행시간에 타입을 확인한후 캐스팅을 수행합니다. RTTI 기술을 사용합니다.
const_cast<>	상수성을 제거할 때 사용합니다.
reinterpret_cast<>	C 의 캐스팅과 유사 합니다. 메모리를 재해석할 때 사용합니다.

캐스팅 분야도 어렵고 복잡한 내용을 많이 담고 있는데. 여기서는 간략하게 살펴 보도록 하겠습니다.

- static_cast

C++ 가장 기본 적인 캐스팅 입니다. 사용방법은 아래와 같습니다. 사용법이 C 캐스팅과 비교해서 약간 복잡해 보입니다.

```
static_cast<변경할 타입>(변수);
```

기본적으로 서로 연관성이 있는 타입의 캐스팅만 허용합니다. 잘못된 결과를 나오게 할 수 있는 캐스팅은 허용되지 않습니다. void*를 다른 타입으로 캐스팅하는 것은 허용됩니다.

```
int main()
{
    int n = 10;
    void* pv = &n; // void*는 모든 주소를 담을수 있습니다.
    int*  pi = static_cast<int*>(pv);   // ok. void*를 다른 타입으로 변경하는 것은
                                        // 프로그램에서 작성시 꼭 필요합니다.

    double* pd = static_cast<double*>(pi); // error. int* => double*로 변경하는 것은
                                           // 위험합니다.
}
```

- reinterpret_cast

static_cast 를 사용하면 int*를 double*로 변경할 수 없습니다. 그러나, 위험 하지만 꼭 필요한 경우가 있을 수 있습니다. 이 경우는 static_cast 가 아닌 reinterpret_cast 를 사용하면 됩니다. reinterpret_cast 는 이름 처럼 메모리를 다시 해석하겠다는 의미이므로, 대부분의 경우 성공합니다.

```
int main()
{
    int n = 10;
    double* p1 = static_cast<double*>(&n);      // error.
    double* p2 = reinterpret_cast<double*>(&n); // ok
}
```

그럼, "reinterpret_cast 를 사용하면 C 처럼 성공하므로 다시 위험해지지 않을까 ?" 라고 생각하시는 분도 계실 겁니다. 그래서, C++은 되도록 이면 reinterpret_cast 를 사용하지 말고 static_cast 만 사용하다가, 컴파일 에러 발생시, 다시 한번 확인해 보고 꼭 필요할 경우에만 reinterpret_cast 를 사용하라는 의도 입니다. C 캐스팅의 경우는 항상 성공하므로 실수로 잘못된 캐스팅을 해도 에러가 나지 않게 됩니다.

- const_cast

const_cast 는 메모리의 상수성을 제거하기 위한 캐스팅 입니다. reinterpret_cast 가 대부분 성공하지만 메모리의 상수성을 제거할 수는 없습니다. 아래 코드를 참고하시면 됩니다.

```
int main()
{
    const int c = 10;   // 상수 입니다.
    int* p1 = static_cast<int*>(&c);        // error. 상수성을 제거할 수 없습니다.
    int* p2 = reinterpret_cast<int*>(&c);   // error. 상수성을 제거할 수 없습니다.
    int* p3 = const_cast<int*>(&c);         // ok. 상수성을 제거할 수 있습니다.
}
```

비록 const_cast 가 메모리의 상수성을 제거해서 const 변수의 주소를 "int*" 형 변수 p 에 담을수 있지만 "*p = 10" 등의 표현을 사용하는 것은 버그의 원인이 될 수 있으므로 사용하지 않은 것이 좋습니다. 단지, 메모리 주소를 잠깐 보관하는 용도로만 사용하시기 바랍니다.

8장. 동적 메모리 할당, nullptr

8.1 동적 메모리 할당(new, delete)

C 언어에서는 동적 메모리 할당을 위해서는 malloc 함수를 사용합니다. 물론, C++에서는 malloc 을 사용할 수 있지만 대부분의 경우는 new 연산자를 사용합니다.

```
int main()
{
    int* p1 = (int*)malloc( sizeof(int) );  // 크기가 인자로 전달됩니다.
    int* p2 = new int;                  // 크기가 아닌 타입을 인자로 전달합니다.

    free(p1);   // malloc 으로 할당한 경우 free로 해지
    delete p2;  // new로 할당한 경우 delete 해지
}
```

malloc 과 new 는 몇가지 차이점이 있습니다.

① **malloc 은 함수 이고 new 는 연산자 입니다.**

② **malloc 은 메모리 크기를 인자로 전달하지만 new 는 type 을 전달합니다.**

③ **malloc 은 메모리 주소를 void* 타입으로 리턴 하기 때문에 캐스팅을 해야 하지만 new 는 해당 타입으로 주소를 리턴 하기 때문에 캐스팅을 할 필요가 없습니다.**

④ **malloc 으로 할당한 메모리는 free 로 해지 하지만, new 로 할당한 메모리는 delete 로 해지 합니다.**

이외에도 malloc 은 생성자를 호출하지 못하지만 new 는 생성자가 호출됩니다. 뒷장에서 생성자를 다룰 때 자세히 살펴 보도록 하겠습니다.

new 는 배열 행태로 메모리 할당도 가능하며 이경우는 메모리를 해지 할 때는 반드시 "delete[]" 를 사용해서 해지 해야 합니다.

```
int* p1 = new int;
delete p1;
```

```
int* p2 = new int[10];  // 배열 형태의 메모리 할당
delete[] p2;            // 반드시 배열 형태로 메모리를 해지 해야 합니다.

int* p3 = new int[10];
delete p3;      // 이렇게 사용하면 안됩니다. 미정의 동작(undefined) 이 발생합니다.
```

8.2 nullptr - C++11

C 언어에서는 0 은 int 타입 이지만 컴파일러에 의해 특별하게 처리 됩니다. 일반적으로 정수는 포인터로 암시적 형 변환 될 수 없지만 0 은 포인터 타입으로 암시적 형변환이 가능합니다.

```
int main()
{
    int  n1 = 10;   // ok.    10은 정수(int) type입니다.
    int* p1 = 10;   // error. 일반적으로 정수타입은 포인터로 암시적 변환 될수 없습니다.

    int  n2 = 0;    // ok
    int* p2 = 0;    // ok. 0은 정수 이지만 포인터로 암시적 형변환 됩니다.
                    // 컴파일러에 의해 특별하게 처리 됩니다.
}
```

포인터를 초기화 하기 위해 오래전부터 0 을 사용해 왔지만, 엄격히 말하면 0 은 포인터 타입은 아니라 int 타입 입니다. 0 이 정수라는 사실은 앞으로 배우게 되는 함수 오버로딩, 함수 템플릿 등에서 문제가 되는 경우가 있습니다. 그래서, C++에서는 (C++11 부터) 포인터 0 을 나타내는 nullptr 을 제공합니다.

```
int main()
{
    // 0은 계속해서, 정수 및 포인터의 초기값으로 사용될 수 있습니다.
    int  n1 = 0;
    int* p1 = 0;

    // 하지만, C++11 부터는 포인터 초기화시 0대신 nullptr을 사용하는 것이 좋습니다.
    int* p2 = nullptr;  // ok
}
```

"0 이 도대체 어떤 문제가 있을까 ?"를 생각하는 독자도 있을 것 같습니다. 0 은 우리가 아직 다루지 않은 템플릿, 함수 오버로딩 등에서 문제가 되는 경우가 있습니다.

nullptr 은 포인터 0(null pointer)를 나타내므로 정수를 초기화 할때는 사용할수 없습니다. 하지만, bool 로의 암시적 형변환은 가능합니다. 단, 직접 초기화시만 가능합니다.

```
int main()
{
    int* p = nullptr;   // ok. 포인터를 초기화 할 때 nullptr을 사용합니다.
    int  n = nullptr;   // error. 정수를 초기화 할때는 nullptr을 사용할수 없습니다.

    bool b1  = nullptr; // error.
    bool b2(nullptr);   // ok. nullptr은 포인터 0 이지만 () 초기화 사용시 bool 타입으로
                        // 암시적 형변환이 허용됩니다.
}
```

[참고] 초기화시 = 를 사용하는 것을 "copy-initialization", ()를 사용하는 것을 direct-initialization" 이라고 합니다. 두 가지 방식의 차이점은 약간 어려운 내용이므로 "C++ Intermediate" 과정을 참고하시기 바랍니다. 또한, VC++ 사용시 "`bool b1 = nullptr`" 코드가 에러가 나오지 않은 경우가 있습니다. 하지만, C++ 표준은 "direct initialization" 을 사용할때만 bool 로 암시적 형변환 가능하다고 되어 있습니다. 이처럼, 컴파일러에 따라 C++ 표준과 약간씩 다르게 동작하는 경우가 있습니다.

[참고] C89 문법에는 bool 타입이 없지만 C99 문법에는 bool 타입이 추가되었습니다. <stdbool.h> 헤더를 포함하면 C 언어 에서도 bool 타입을 사용할수 있습니다.

C/C++ 에서 모든 값은 타입이 있습니다. "1"은 int 타입이고, "3.4" 는 double 타입 입니다. nullptr 도 nullptr_t 타입 입니다. 또한, nullptr_t 를 사용하려면 <utility> 헤더가 포함되어야 합니다.

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

int main()
{
    int     n = 1;      // 1  은 int 타입 입니다.
    double  d = 3.4;    // 3.4는 double 타입입니다.
    char    c = 'A';    // 'A'는 char 타입입니다.

    nullptr_t t = nullptr;  // nullptr은 nullptr_t 타입입니다.
    int* p = t;             // t는 결국 nullptr와 동일합니다.
}
```

[참고] nullptr 이 nullptr_t 타입이라는 사실을 정확히 이해 하려면 C++의 다른 문법을 좀더 배워야 합니다. 아직 많은 내용을 다루지 않았으므로 현재 까지는 nullptr 만 알고 있어도 충분합니다.

SECTION 2

Object Oriented Programming

C++은 기존의 C 에 있던 함수, 변수를 보다 사용하기 쉽게 발전시킨 다양한 문법을 제공합니다.

하지만 역시 C++의 가장 중요한 특징은 객체 지향 프로그래밍을 위한 클래스 문법 입니다.

2 장에서는 C++이 지원 하는 다양한 클래스(class) 문법에 대해 살펴 보도록 하겠습니다.

이번 장에서 다음과 같은 개념을 배우게 됩니다.

"객체 지향 프로그래밍 개념, 접근지정자/생성자/소멸자, 초기화 리스트(initialize list), 복사 생성자(copy constructor), 객체 복사(object copy), 정적 멤버(static member),상수 멤버 함수(const member function), this pointer"

9장.Object Oriented Programming.

9.1 객체지향 프로그래밍 개념

함수 오버로딩, 레퍼런스, auto, ragned-for 등 C++에는 C 에는 없는 많은 요소를 추가로 제공합니다. 하지만, C++의 가장 큰 특징은 역시 "객체지향 프로그래밍" 과 "일반화 프로그래밍" 의 개념입니다. 이번 장에서는 C++의 가장 큰 핵심인 "객체지향 프로그래밍"에 대해서 살펴 보도록 하겠습니다.

간단한 예제를 가지고 객체 지향 프로그래밍의 개념에 대해서 살펴 보도록 하겠습니다.

▪ 복소수 2 개의 합을 구하는 함수 만들기

복소수 2 개의 합을 구하는 함수를 생각해 봅시다. 먼저 복소수는 실수부와 허수부가 있으므로 2 개의 변수를 사용해야 한 개의 복소수를 표현 할 수 있습니다. 결국 2 개의 복소수를 전달 받으려면 4 개의 파라미터가 필요합니다. 그런데, 문제가 되는 것은 C 의 함수는 리턴 값을 하나밖에 만들 수 없다는 점입니다.

```
? Add(int ar, int ai, int br, int bi)
{
    int sum_r = ar + br;
    int sum_i = ai + bi;

    return ? ; // C 에서 함수는 한개의 값만 리턴 할 수 있습니다.
}
int main()
{
    int ar = 1, ai = 1; // 1 + 1i
    int br = 2, bi = 2; // 2 + 2i

    ? ret = Add(ar, ai, br, bi);
}
```

결국, 위 코드에서 복소수 하나는 int 타입의 변수 2 개로 표현 되고 있으므로 Add 함수에서 값을 2 개를 리턴 해야 합니다. 어떻게 해야 할까요 ?

▪ Out Parameter 를 사용해서 값 꺼내 오기

함수의 인자로 변수의 주소를 보내면 함수로부터 값을 2 개이상 꺼내 올 수 있습니다. 다음 코드에서 Add 함수의 5, 6 번째 파라미터를 생각해 봅시다.

```
void Add(int ar, int ai, int br, int bi, int* sr, int* si)
{
    // 5, 6번재 인자로 전달된 변수에 결과값을 담아 줍니다.
    *sr = ar + br;
    *si = ai + bi;
}
int main()
{
    int ar = 1, ai = 1;
    int br = 2, bi = 2;
    int sr = 0, si = 0; // 결과 값을 담아오기 위한 변수

    Add(ar, ai, br, bi, &sr, &si);
}
```

위 코드에서 Add 함수의 5 번째, 6 번째 파라미터는 값을 전달하기 위한 목적이 아니라 함수로부터 값을 꺼내 오기 위한 목적으로 사용됩니다. 이런 파라미터를 "Out Parameter"라고 부릅니다.

결국, 함수로부터 값 2 개를 꺼내 오는 데는 성공했습니다. 그런데, 코드가 약간 복잡해 보입니다. 왜 복잡해 보일까요 ?

위 코드가 복잡해 보이는 이유는

"우리는 복소수를 다루는 프로그램을 만들고 싶은데 C 언어에는 복소수라는 타입이 없기 때문에 int 타입의 변수 2 개를 사용해서 복소수를 표현했습니다. 그래서, 코드가 복잡해 보입니다."

만약 C 언어에 "복소수"라는 타입이 있으면 코드는 훨씬 단순해 보일 수 있습니다."

- Complex 타입 만들기

C 언어에서도 구조체를 사용하면 새로운 타입을 만들 수 있습니다. 구조체를 사용해서 프로그램에서 필요한 Complex 라는 타입을 먼저 만들고 Add 함수를 만들면 프로그램은 좀더 간단해 보일 수 있습니다.

```
// "Complex" 타입을 먼저 설계합니다.
struct Complex
{
```

```
    int re;
    int im;
};
// "Complex" 2개를 인자로 받아서 Complex 를 리턴 하는 함수 입니다.
Complex Add(const Complex& c1, const Complex& c2)
{
    Complex ret = { c1.re + c2.re, c1.im + c2.im };
    return ret;
}
int main()
{
    Complex c1 = { 1,1 };
    Complex c2 = { 2,2 };

    Complex ret = Add(c1, c2);
}
```

이 처럼 프로그램에서 필요한 타입을 먼저 설계하고 타입을 사용해서 프로그램을 만들면 훨씬 쉽게 프로그램을 만들 수 있습니다. 결국, 객체지향 프로그램은 프로그램에서 필요한 타입(객체)를 먼저 설계하고 프로그램을 만들자는 개념입니다.

그럼, C 의 구조체를 사용해도 충분한데, 왜 C++을 사용할까요 ? C 의 구조체는 데이터만 넣을 수 있지만 C++의 구조체는 데이터 뿐 아니라 생성자, 소멸자, 멤버 함수, 접근 지정자 등을 추가로 넣을 수 있습니다. 다른 예제를 가지고 좀더 자세히 살펴 보도록 하겠습니다.

참고로

"int n" 에서 흔히 n 을 변수라고 부릅니다. 하지만, Complex 같은 사용자 정의 타입의 변수 c 는 "변수"라는 용어 보다는 **"객체"** 라는 용어를 더 많이 사용합니다. 앞으로는 "변수" 라는 용어 대신 "객체"라는 용어를 사용하겠습니다.

9.2 Stack 만들기로 배우는 객체지향 프로그래밍

스택은 마지막에 넣은 데이터가 먼저 나오게 되는 "후입 선출(Last In First Out)" 방식의 자료구조 입니다. 이번에는 단계별로 스택을 만들어 가면서 객체지향 프로그래밍을 위한 C++의 다양한 문법을 배워 보도록 하겠습니다. 한가지 문법을 깊이 있게 살펴 보기 보다는 먼저 전체적인 개념을 살펴 보도록 하겠습니다. 각 단계에서 사용된 문법에 뒷장에서 다시 자세히 살펴 보도록 하겠습니다.

Step1. 전역변수와 함수를 사용한 Stack

첫번째 방법은 전역변수와 일반 함수를 사용한 간단한 모양입니다. 어렵지 않으므로 쉽게 이해 할 수 있는 코드 입니다.

```
int buff[10] = { 0 };
int idx = 0;

void push(int v)
{
    buff[idx++] = v;
}
int pop()
{
    return buff[--idx];
}
int main()
{
    push(10);
    push(20);

    cout << pop() << endl;
    cout << pop() << endl;
}
```

위 코드의 단점 중의 하나는 스택이 2 개 이상 필요한 경우 전역 변수와 함수를 계속 추가해야 합니다.

Step 2. Stack 타입의 설계

이번에는 구조체를 사용해서 Stack 타입을 먼저 설계한 후 사용해 보도록 하겠습니다. 역시, 어렵지 않은 간단한 코드 입니다.

```
struct Stack
{
    int buff[10];
    int idx;
};
void push(Stack* s, int v)
{
    s->buff[(s->idx)++] = v;
}
int pop(Stack* s)
{
    return s->buff[--(s->idx)];
}
int main()
{
    Stack s1, s2; //  2개 이상의 스택을 만들 수 있습니다.
    s1.idx = 0;
    s2.idx = 0;    //  2개의 스택을 각각 초기화 하고 사용하면 됩니다.
    push(&s1, 10);
    push(&s1, 20);

    cout << pop(&s1) << endl;
    cout << pop(&s1) << endl;
}
```

위 코드는 구조체를 사용해서 Stack 타입을 먼저 설계 한 후에 사용하고 있습니다. Stack 이 2 개 이상 필요한 경우에도 main 함수 처럼 Stack 형 변수를 2 개 만들어서 init 함수로 초기화 한 후에 사용하면 됩니다.

이 코드의 단점은 Stack 관련된 관련된 모든 함수가 반드시 Stack 변수를 인자로 받아야 한다는 점입니다. 역시 코드가 복잡해 보입니다.

```
//  스택 관련 모든 함수가 1번째 인자로  Stack 변수를 전달받고 있습니다.
void push(Stack* s, int v) { s->buff[(s->idx)++] = v;}
int pop(Stack* s)          { return s->buff[--(s->idx)];}
```

위와 같은 코드가 나오는 이유는 Stack 이라는 타입을 만들 때, 상태를 나타내는 Data 만 가지고 있고, Stack 을 조작하는 함수는 별도의 일반 함수로 되어 있기 때문 입니다.

만약, Stack 을 만들 때 상태를 나타내는 데이터 뿐 아니라 스택을 조작하는 함수도 구조체 안에 넣을 수 있다면 어떨까요 ?

Step 3. 상태를 나타내는 데이터와 상태를 조작하는 함수를 묶기

C 언어 에서는 구조체 안에 데이터만 넣을 수 있지만 C++에서는 구조체 안에 함수도 넣을 수 있습니다. 구조체 안에 있는 데이터를 멤버 데이터, 함수를 멤버 함수라고 부르는데, 멤버 함수 안에서는 멤버 data 에 마음데로 접근 할 수 있습니다.

```
struct Stack
{
    // 멤버 데이타
    int buff[10];
    int idx;

    // 멤버 함수 안에서는  멤버 데이타에 접근할수
    // 있으므로 함수 인자로 Stack을 받을 필요가 없습니다.
    void push(int v)
    {
        buff[idx++] = v;
    }
    int pop()
    {
        return buff[--idx];
    }
};
int main()
{
    Stack s1;
    s1.idx = 0;
    s1.push(10);
    s1.push(20);
    cout << s1.pop() << endl;
    cout << s1.pop() << endl;
}
```

아래 코드를 보고 함수가 외부에 있을 때와 구조체 내부에 있을 때의 차이점은 잘 생각해 봅시다.

```
push(&s1, 10);  // 함수가 구조체 안에 있지 않은 경우
                // 함수의 인자로 객체를 전달합니다. 함수 중심입니다.

s1.push(10);    // 함수를 구조체 안에 포함한 경우.
                // 객체가 가진 함수를 사용합니다. 객체 중심 입니다.
```

C 언어의 구조체와 C++의 구조체의 가장 큰 차이점 중의 하나는 함수를 포함할 수 있다는 점입니다. 이제 프로그래밍의 중심이 함수 중심에서 객체 중심이 되었습니다.

이 코드는 어떤 단점이 있을까요 ? 지금까지 만든 Stack 을 사용하려면 반드시 멤버 data 인 idx 를 0 으로 초기화 하고 사용해야 합니다. 그런데, 사용자가 사용하다가 아래 처럼 idx 를 변경하면 어떻게 될까요 ?

```
int main()
{
    Stack s1;
    s1.idx = 0;
    s1.push(10);
    s1.idx = 10; // 사용자가 실수를 했습니다. 더 이상 s1을 사용할 수 없습니다.
    s1.push(20);
}
```

그런데, 생각해 볼 점은 사용자 입장에서는 Stack 에 넣었다 빼는 push()/pop() 멤버 함수만 알면 되지, Stack 이 내부적으로 idx 와 buff 배열로 data 를 관리한다는 사실은 알 필요가 없습니다.

Stack 의 상태를 나타내는 buff 배열과 idx 멤버는 Stack 사용자가 접근 할 수 없게 하는 것이 좋습니다.

Step 4. 정보 은닉 (Information Hiding) - 접근 지정자 도입

일반 적으로 Stack 의 사용자는 data 를 넣었다 빼는 push/pop 함수만 알면 되지, buff 배열과 idx 멤버를 알 필요가 없습니다. 오히려 idx 를 접근 할 수 있게 되면 잘못된 값을 넣어서 객체의 상태가 불안정해 질 수 있습니다.

private 키워드를 사용하면 특정멤버를 외부에서 접근 할 수 없게 할 수 있습니다.

```
struct Stack
{
private:            // 이 영역에 있는 멤버는 외부에서 접근할 수 없습니다.
    int buff[10];   // 멤버 함수에서만 접근할 수 있습니다.
    int idx;        // 다른 접근지정자가 나올때 까지 private이 계속됩니다.

public:             // 이 아래 부분에 있는 멤버는 외부에서 접근할 수 있습니다.
    void init() {
        idx = 0;
    }
    void push(int v) {
        buff[idx++] = v;
    }
    int pop() {
        return buff[--idx];
    }
};
int main()
{
    Stack s1;
    //s1.idx = 0; // error. private 멤버를 외부에서 접근할 수 없습니다.
    s1.init();  // idx를 초기화 하기 위해 멤버 함수를 사용해야 합니다.
    s1.push(10);
    s1.push(20);
}
```

이제는, Stack 을 사용하는 사람은 idx 를 접근할 수 없게 되었습니다. 사용자가 idx 에 잘못된 값을 넣는 실수를 할 수 없게 되었습니다. 대신에 사용자는 stack 을 초기화 하기 위해 반드시 init 함수를 호출해야 합니다.

만약 접근지정자를 생략한다면 어떻게 될까요 ? 접근 struct 는 접근지정자를 표시하지 않으면 public 입니다. 하지만, C++에서 새로 도입된 class 라는 키워드를 사용하면 접근지정자 생략시 private 이 디폴트가 됩니다.

```
struct Stack
{
    int idx;    // 접근지정자를 표시하지 않으면 public 입니다.
};
class Stack
{
    int idx;    // 접근지정자를 표시하지 않으면 private 입니다.
};
```

[참고] struct 와 class 의 차이점은 접근 지정자 생략시 디폴트 값만 차이가 있고 나머지는 거의 동일합니다.

결국, class 는 기본적으로 접근을 막아 놓고 필요한 경우만 public 키워드를 사용해서 접근하게 하자는 의미 입니다. 이제부터는 struct 대신 class 를 사용하도록 하겠습니다.

현재 까지 Stack 의 단점은 Stack 을 사용하기 위해서는 항상 init 함수를 호출해서 먼저 초기화 하고 사용해야 합니다.

Stack 을 자동으로 초기화되게 할 수 없을까요 ?

Step 5. 생성자(Constructor) 도입

클래스(또는 struct) 이름과 동일한 함수를 생성자(Constructor)라고 합니다.

```
class Stack
{
    // ……
public:
    // 생성자 - 함수 이름이 클래스 이름과 동일하고 리턴 타입을 표기하지 않습니다.
    Stack() {
    }
};
```

생성자는 다음과 같은 특징이 있습니다.

- **함수 이름이 클래스이름과 동일하다**
- **리턴 값이 없고 리턴 타입 자체를 표기하지 않는다.**
- **인자는 있어도 되고 없어도 된다.**

- **객체를 생성하면 자동으로 호출된다.**

Stack 의 초기화 작업을 init 에서 하지 말고 생성자에서 하면 Stack 객체 생성시 자동으로 생성자가 호출되므로 Stack 을 초기화 할 수 있게 됩니다.

```
class Stack
{
private:
    int idx;
    int buff[10];
public:
    Stack() {
        idx = 0;
    }
    void push(int v) {
        buff[idx++] = v;
    }
    int pop() {
        return buff[--idx];
    }
};
int main()
{
    Stack s1;    // 이순간 생성자가 호출되어서 idx 가 0으로 초기화 됩니다.
    s1.push(10);
    s1.push(20);
}
```

이제 생성자 덕분에 Stack 객체를 만들기만 하면 init 함수를 호출할 필요 없이 자동으로 초기화 되므로 사용자 입장에서는 훨씬 편리해 졌습니다.!

그럼, 이제 어떤 문제가 남아 있을까요 ? 현재 Stack 은 내부적으로 10 개 짜 리 배열을 사용하고 있습니다. 따라서 사용자가 10 개 이상의 data 를 넣을 경우 문제가 생기게 됩니다.

Stack 사용자가 내부 버퍼의 크기를 결정할 수 있게 하면 어떨까요 ?

Step 6. 자료 구조의 변경과 소멸자(Destructor) 도입

생성자는 인자를 가져도 되고 인자를 같지 않아도 됩니다. 그리고, 여러 개를 제공해도 됩니다.

```
class Stack
{
    // ……
public:
    Stack() {           // 인자가 없는 생성자
    }
    Stack(int sz) {     // 인자가 한 개인 생성자
    }
};
int main()
{
    Stack s1;       // 인자가 없는 생성자가 호출됩니다.
    Stack s2(100);  // 인자가 하나인 생성자가 호출됩니다.
}
```

이번에는 Stack 사용자가 버퍼의 크기를 생성자로 전달해서 동적 메모리 할당을 할 수 있도록 변경해 보겠습니다.

```
class Stack
{
private:
    int idx;
    int* buff; // 배열이 아닌 포인터를 사용합니다.
public:
    Stack(int sz = 10)
    {
        idx = 0;
        buff = new int[sz];// 사용자가 전달한 크기 만큼 버퍼의 할당합니다.
    }
    // ……
};
int main()
{
    Stack s1(100);  // 인자가 하나인 생성자가 호출됩니다.
}
```

이제, Stack 사용자가 전달한 크기 만큼의 버퍼를 만들어서 사용하게 됩니다. 만약, 전달하지 않으면 디폴트 값인 10 만큼의 메모리를 할당하게 됩니다. 그런데, 남은 문제는 동적 메모리 할당을 했으므로 Stack 이 더 이상 필요 없을 때 반드시 메모리를 해지 해야 합니다.

객체를 만들면 생성자가 호출되는 것처럼 객체가 파괴 될 때는 소멸자가 호출됩니다. 생성자는 클래스 이름과 동일한 이름으로 만들지만 소멸자는 "~클래스이름()"을 사용해서 만들게 됩니다.

```
class Stack
{
public:
    // ……
    Stack(int sz = 10) {    // 생성자 – 객체가 생성될 때 호출됩니다.
    }
    ~Stack() {              // 소멸자 – 객체가 파괴될 때 호출됩니다.
    }
};
```

소멸자(Destructor)는 다음과 같은 특징이 있습니다.

- **함수 이름은 "~클래스이름()" 입니다.**
- **리턴 타입을 표기하지 않습니다.**
- **인자를 가질 수 없습니다.**
- **객체가 파괴 될 때 호출됩니다.**

완성된 코드는 다음과 같습니다.

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

class Stack
{
private:
    int idx;
    int* buff;
public:
    Stack(int sz = 10) {
        idx = 0;
        buff = new int[sz];
    }
```

```
    ~Stack() {
        delete[] buff;
    }
    void push(int v) {
        buff[idx++] = v;
    }
    int pop() {
        return buff[--idx];
    }
};
int main()
{
    Stack s1(100);
    s1.push(10);
    s1.push(20);
    cout << s1.pop() << endl;
    cout << s1.pop() << endl;
}
```

Step 7. 선언과 구현의 분리

C 언어 에서 함수를 만들 때 선언과 구현부를 분리 하듯이 C++ 클래스도 클래스 안에는 함수 선언만 놓고 구현을 클래스 밖에 구현하는 것이 원칙입니다.

멤버 함수를 클래스 외부에 구현 할 때는 "클래스이름::함수이름()"의 형식으로 구현 합니다.

```
class Stack
{
public:
    // ……
    void push(int v); // 클래스 안에는 멤버 함수의 선언만 제공합니다.
};
// 멤버함수를 클래스 외부에 구현합니다.
void Stack::push(int v)
{
    buff[idx++] = v;
}
```

또한, 클래스 선언부는 헤더 파일(.h)에 놓고, 멤버 함수 구현부는 소스파일(.cpp)로 만들게 됩니다. 또한, Stack 사용자는 헤더만 포함해서 사용하면 됩니다.

파일별로 분할된 완전한 모양의 코드 입니다. 컴파일 할 때는 g++과 cl (MS 컴파일러)

```
g++ main.cpp Stack.cpp // 또는   cl main.cpp Stack.cpp
```

로 컴파일 하면 됩니다.

Stack.h

```
#ifndef _STACK_H_
#define _STACK_H_

class Stack
{
private:
    int idx;
    int* buff;
public:
    Stack(int sz = 10);
    ~Stack();
    void push(int v);
    int pop();
};

#endif // _STACK_H_
```

Stack.cpp

```
#include "Stack.h"

Stack::Stack(int sz /* = 10 */ ) {
    idx = 0;
    buff = new int[sz];
}
Stack::~Stack() {
    delete[] buff;
}
void Stack::push(int v) {
    buff[idx++] = v;
}
int Stack::pop() {
    return buff[--idx];
}
```

```
// main.cpp

#include <iostream>
#include "Stack.h"
using namespace std;

int main()
{
    Stack s(100);
    s.push(10);
    s.push(20);

    cout << s.pop() << endl;
```

```
    cout << s.pop() << endl;
}
```

Step 8. 코딩 관례

Stack 의 사용자 입장에서는 Stack 의 상태를 나타내는 멤버 data 를 알 필요가 없습니다. push/pop 함수만 알면 stack 을 사용할 수 있습니다. 그래서 보통 Stack 클래스의 선언부를 만들 때 public 멤버 함수를 먼저 선언하고 private 으로 되어 있는 아래 부분에 놓는 경우가 많이 있습니다.

멤버 데이터를 먼저 선언한 경우

```
class Stack
{
private:
    int idx;
    int* buff;
public:
    Stack(int sz = 10);
    ~Stack();
    void push(int v);
    int pop();
};
```

멤버 함수를 먼저 선언한 경우

```
class Stack
{
public:
    Stack(int sz = 10);
    ~Stack();
    void push(int v);
    int pop();
private:
    int idx;
    int* buff;
};
```

[참고] 반드시 멤버 함수를 먼저 선언해야 하는 것은 아닙니다. 또한, 멤버 데이터를 먼저 선언하는 경우도 많이 있습니다.

Step 9. 클래스 템플릿 (template)

지금 까지 만든 Stack 의 또 다른 문제점은 int 타입만 저장할 수 있다는 점입니다. 실수를 저장하는 Stack 을 만들려면 buff 를 int* 가 아닌 double*로 해야 합니다. 물론, push/pop 함수의 인자와 리턴 타입도 double 로 변경해야 합니다.

하지만, 타입만 변경하면 나머지 구현은 완전히 동일하게 됩니다. 타입만 다르고 구현이 동일할 때 함수 템플릿을 사용하듯이 클래스도 템플릿으로 만들 수 있습니다.

이때, 주의 할 점은 클래스를 템플릿으로 만들 때는 헤더 파일과 구현 파일로 분리하면 안되고 반드시 하나의 파일(헤더파일)안에 선언과 구현부를 같이 제공해야 합니다.

StackT.h

```
#ifndef _STACK_T_H_
#define _STACK_T_H_

// int 대신 T를 사용합니다.
template<typename T> class Stack
{
public:
    Stack(int sz = 10) {
        idx = 0;
        buff = new T[sz];
    }
    ~Stack() {
        delete[] buff;
    }
    void push(T v) {
        buff[idx++] = v;
    }
    T pop() {
        return buff[--idx];
    }
private:
    int idx;
    T* buff;
};
#endif // _STACK_T_H_
```

main.cpp

```
#include <iostream>
#include "StackT.h"
using namespace std;

int main()
{
    Stack<double> s(100);
    s.push(1.1);
    s.push(2.2);

    cout << s.pop() << endl;
    cout << s.pop() << endl;
}
```

[참고] 클래스 템플릿을 만들 때 멤버 함수의 구현을 클래스 외부에 할 수도 있습니다. 단, 이경우도 외부 구현도 반드시 헤더 파일안에 있어야 합니다.

Step10. STL 의 Stack

우리는 지금까지 Stack 을 직접 만들었지만 C++ 표준 라이브러리인 STL 에는 이미 stack 을 제공하고 있습니다. STL 의 stack 은 다음과 같은 특징이 있습니다

- **<stack> 헤더를 포함해야 합니다.**
- **사용자가 버퍼 크기를 전달 할 필요 없습니다.**
- **pop 함수는 제거만 하고 리턴을 하지 않습니다. 값을 꺼낼 때는 top 멤버 함수를 사용합니다. 단, top 은 제거하지 않습니다.**

아래 코드는 STL 의 stack 을 사용하는 간단한 코드 입니다.

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

int main()
{
    stack<int> s;

    s.push(10);
    s.push(20);
    //int n = s.pop(); // error. pop()은 제거만 하고 리턴 하지 않습니다.
    cout << s.top() << endl; // 20. top은 제거하지 않습니다
    cout << s.top() << endl; // 20
    s.pop();
    cout << s.top() << endl;
}
```

지금 까지 Stack 만들기를 통해서 객체지향 프로그램의 개념을 살펴 보았습니다. 이제, 각 문법에 대해서 좀더 자세히 살펴 보도록 하겠습니다.

10장.접근지정자, 생성자 소멸자

10.1 접근지정자, 정보 은닉, 캡슐화

private 접근지정자를 사용하면 외부에서는 접근 할 수 없고 멤버함수에서만 접근가능한 멤버를 만들 수 있습니다.

```
class Stack
{
private:
    int idx;
public:
    Stack() { idx = 0; } // 멤버 함수에서는 private 멤버를 접근할 수 있습니다.
};
int main()
{
    Stack s;
    s.idx = 10; // error. 멤버함수가 아닌 곳 에서는 private 멤버에 접근 할 수 없습니다.
                // 사용자는 idx에 잘못된 값을 넣을 수 없습니다.
}
```

장점은, 외부의 잘못된 사용으로부터 객체(Stack)의 상태(idx) 가 잘못되는 것을 막을 수 있습니다. 이제 사용자가 아무리 Stack 의 사용법을 잘 몰라도, Stack 의 상태가 불안해지지 않습니다. 이처럼 객체의 상태를 나타내는 data 를 private 영역에 놓아서 외부의 잘못된 사용으로부터 객체를 안전하게 하는 것은 **"정보은닉(information Hiding)"** 이라고 합니다. 또한, 멤버 데이터를 외부에 직접 노출하지 않고 멤버 함수를 통해서만 접근하게 되어 있으므로 캡슐처럼 되어 있다는 의미로 **"캡슐화(encapsulation)"** 이라고도 합니다.

java 나 C#에서는 각 멤버 별로 각각 접근 지정자 키워드를 각각 적어야 합니다. 하지만 C++에서는 한번 적으면 새로운 접근 지정자가 나올 때 까지 계속 유지 됩니다.

```
// java, C# 스타일
class Stack
{
    private int idx;
    private int buff;
};
```

```
// C++ Style
class Stack
{
private:
    int idx;
    int* buff;
    // 새로운 접근 지정자가 나올 때 까지 계속 private 입니다.
};
```

물론 C++에서도 접근 지정자를 여러 번 적어도 전혀 문제가 없고, 멤버 함수를 private 으로 만들어도 됩니다.

```
class Stack
{
private:
    int idx;
private:        // 접근지정자를 여러 번 표기해도 상관없습니다.
    int* buff;
};
```

- **friend 함수**

멤버가 아닌 일반 함수에서 private 멤버를 접근할 수는 없습니다. 하지만 friend 함수로 선언하면 멤버 함수가 아니어도 private 멤버에 접근할 수 있습니다.

```
class Stack
{
private:
    int idx;
public:
    Stack() { idx = 0; }

    friend void foo();
};
void foo()
{
    Stack s;
```

```
    s.idx = 0;  // ok. friend 함수에서는 private 멤버를 접근할 수 있습니다.
}
```

또한, friend 함수의 만들 때 구현부를 클래스 선언 안에 만들 수도 있습니다.

```
class Stack
{
private:
    int idx;
public:
    Stack() { idx = 0; }

    // 아래 foo 함수는 멤버 함수가 아닙니다. 일반 함수 입니다.
    friend void foo()
    {
        Stack s;
        s.idx = 0;
    }
};
```

friend 함수를 클래스 선언 안에 구현하는 방식은 클래스 템플릿에서 널리 사용되는 기법 중 하나입니다.

friend 함수 뿐 아니라 friend class 도 만들 수 있습니다.

```
class Stack
{
private:
    int idx;
public:
    Stack() { idx = 0; }
    friend class Calc;  // Calc의 모든 멤버함수에서 private멤버에 접근할 수 있습니다.
};
class Calc
{
public:
    void useStack()
    {
        Stack s;
```

```
        s.idx = 0;      // ok
    }
};
```

왜, 언제 friend 함수가 필요한가? friend 함수는 "정보은닉"과 "캡슐화" 를 위배 하는 것이 아닌가 ? 라는 의문이 드시는 분도 있을 겁니다. 앞으로 연산자 재정의 등을 배울 때 friend 함수를 자주 사용하게 되므로 그 때 자세히 설명하도록 하겠습니다. 지금은 문법만 정확히 알아 두시면 됩니다.

10.2 생성자 (constructor)

객체를 만들면 자동으로 생성자가 호출됩니다. 생성자는 다음과 같은 특징이 있습니다.

✓ **클래스 이름과 동일하며 리턴 값을 가질 수 없고, 리턴 타입을 표기하지도 않습니다.**

✓ **생성자를 2 개 이상 제공 할 수도 있습니다.**

✓ **사용자가 만들지 않으면 컴파일러가 인자가 없는 생성자를 하나 제공 합니다. (디폴트 생성자)**

```
class Point
{
private:
    int x;
    int y;
public:
    Point()             { x = 0; y = 0; }   // 1
    Point(int a, int b) { x = a; y = b; }   // 2
};
int main()
{
    Point p1;       // 1번 생성자를 호출합니다
    Point p2(1, 2); // 2번 생성자를 호출합니다.
    Point p2(1);    // error. 인자가 1개인 생성자는 없습니다.
}
```

지금부터 생성자 관련 문법에 대해서 자세히 살펴 보도록 하겠습니다.

- **default 생성자**

사용자가 생성자를 한 개도 만들지 않으면 컴파일러가 인자가 없는 생성자를 한 개를 제공해 줍니다.

```
class Point
{
    int x;
    int y;

    // 사용자가 만든 생성자가 없습니다.
    // 컴파일러가 인자가 없는 생성자 하나를 제공해줍니다
    // Point() {}
```

```
};
int main()
{
    Point p1;       // ok. 컴파일러가 제공해주는 default 생성자를 사용합니다.
    Point p2(1, 2); // error. 인자가 2개인 생성자는 없습니다.
}
```

물론 사용자가 생성자를 한 개 라도 제공하면 컴파일러는 디폴트 생성자를 제공하지는 않습니다.

```
class Point
{
    // ……
public:
    Point(int a, int b) { x = a; y = b; }
};
int main()
{
    Point p1;       // error. 사용자가 생성자를 제공하면 컴파일러는
                    // default 생성자를 제공하지 않습니다.
    Point p2(1, 2); // ok.
}
```

- **배열과 생성자**

객체를 1 개 만들면 생성자는 한번 호출되고, 객체를 2 개 만들면 생성자는 2 번 호출되게 됩니다. 객체 배열을 만들면 배열이 가진 요소가 생성될 때 생성자가 호출되어야 합니다. 디폴트 생성자가 없는 객체의 경우 배열을 만들 때 주의 해야 합니다.

```
int main()
{
    Point p1[3]; // 디폴트 생성자 가 3번 호출됩니다.
                 // Point의 디폴트 생성자가 없다면 에러가 나옵니다.

    Point p2[3] = { Point(0,0), Point(0,0) };   // 인자가 2개인 생성자가 2번,
                                                // 디폴트 생성자가 1번 호출됩니다.
}
```

- **new 와 생성자**

객체를 힙 에 만들 때 new 는 생성자를 호출하지만, malloc 은 생성자가 호출되지 않습니다.

```
int main()
{
    Point* p1; // 객체를 만든 것이 아닙니다. 생성자가 호출되지 않습니다.

    Point* p2 = new Point; // 디폴트 생성자가 호출됩니다.
    Point* p3 = new Point(0, 0); // 인자가 2개인 생성자가 호출 됩니다.

    // Point 객체를 만든것이 아닙니다.
    // Point 크기의 메모리를 할당한 것입니다. 생성자가 호출되지 않습니다.
    Point* p4 = static_cast<Point*>(malloc(sizeof(Point)));

    free(p4); // 소멸자가 호출되지 않습니다

    delete p3;
    delete p2; // 소멸자가 호출됩니다.
}
```

- **멤버의 생성자가 먼저 호출된다.**

객체를 생성하면 자신의 생성자 뿐 아니라 멤버의 생성자도 호출되게 됩니다. 호출순서는

① **멤버의 생성자가 먼저 호출되고**

② **자신의 생성자가 호출됩니다.**

```
class Rect
{
    Point p1;
    Point p2;
public:
    Rect() { cout << "Rect()" << endl; }
};
int main()
{
    Rect r; // 이순간
            // (1) 멤버 데이터 p1의 생성자 호출
```

```
            // (2) 멤버 데이터 p2의 생성자 호출
            // (3) Rect 자신의 생성자 호출
}
```

10.3 소멸자 (destructor)

모든 객체는 생성될 때 생성자가 호출되고, 파괴 될 때 소멸자가 호출됩니다. 소멸자는 다음과 같은 특징이 있습니다.

- ✓ **함수 이름은 "~클래스이름()" 입니다.**
- ✓ **리턴 타입을 표시하지 않고, 인자도 가질 수 없습니다.**
- ✓ **인자가 없으므로 함수 오버로딩이 불가능하고 오직 한 개만 만들 수 있습니다.**
- ✓ **생성자에서 자원을 획득한 경우, 소멸자에서 자원을 반납해야 합니다.**
- ✓ **사용자가 소멸자를 만들지 않을 경우 컴파일러가 제공해 줍니다.**

```
class Point
{
    int x, int y;
public:
    Point() { cout << "Point()" << endl; }

    // 소멸자는 오직 한 개 밖에 만들 수 없습니다
    ~Point() { cout << "~Point()" << endl; }
};
int main()
{
    Point p1;   // 생성자 호출
}   // 블럭을 벗어날 때 p1이 파괴 되고 소멸자가 호출됩니다.
```

- **소멸자와 자원 관리**

일반적으로 생성자에서 자원을 할당한 경우 소멸자에서 자원을 반납해야 합니다.

```
class People
{
    char* name;
public:
    People(const char* n)
    {
        name = new char[strlen(n) + 1]; // 생성자에서 메모리 할당
```

```
        strcpy(name, n);
    }
    ~People()
    {
        delete[] name;  // 소멸자에서 메모리 해지
    }
};
```

- **private 소멸자**

소멸자는 비교적 쉬운 문법 이므로 이해하기는 어렵지 않습니다. 소멸자의 마지막 이야기로 간단한 테크닉을 하나 소개하고 마무리 하도록 하겠습니다.

소멸자를 private 영역에 만들면 어떻게 될까요 ? 객체가 파괴 되려면 소멸자를 호출해야 하는데, 소멸자가 private 영역에 있으므로 호출할 수 없습니다. 따라서, 객체를 생성하는 코드를 만들 때 컴파일 에러가 나오게 됩니다.(정확히는 소멸자를 호출할 수 없기 때문에 에러입니다.)

```
class OnlyHeap
{
private:
    ~OnlyHeap() {}
};

int main()
{
    OnlyHeap h1; // 소멸자를 호출할 수 없으므로 객체를 만들 수 없습니다.

    OnlyHeap* p = new OnlyHeap; // 이 순간은 에러가 아닙니다.
    delete p;                   // 이 순간 소멸자를 호출할 수 없으므로 에러입니다.
}
```

결국, 스택과 힙에 모두 객체를 만들 수 없는데, 왜 소멸자를 private 에 놓았을까요 ?

소멸자가 private 에 있으면 스택에 객체를 만들 수 는 없지만 힙에 객체를 만들 수 있습니다. 객체가 자기 자신을 스스로 delete 할 수 있도록 멤버 함수를 제공하면 됩니다.

```
class OnlyHeap
{
private:
```

```
    ~OnlyHeap() {}
public:
    void Destroy(OnlyHeap* p)
    {
        delete p; // 멤버 함수에서는 private 소멸자를 호출할 수 있습니다.
    }
};
int main()
{
    OnlyHeap* p = new OnlyHeap;
    p->Destroy(p);          // ok!
}
```

[참고] 위코드에서는 Destroy()가 인자로 객체의 주소를 받고 있습니다. 하지만, 뒤에 배우는 this 를 사용하면 좀더 편리하게 코드를 작성할 수 있습니다.

11장. 초기화 리스트(initializer list)

11.1 멤버 데이터 초기화 방법

C++에서는 생성자에서 멤버 data 를 초기화 할 때는 2 가지 방법이 있습니다.

① 생성자의 함수 블록 { } 안에서 멤버 데이터를 초기화 하는 방법

② 생성자의 () 뒤에 콜론(:) 을 적고 멤버 data 를 초기화 하는 방법 – 초기화 리스트(initialize list)

이중에서 생성자 뒤에 : 을 적고 멤버를 초기화하는 것을 **"초기화 리스트(initialize list)"** 라고 합니다. 아래 예제 코드가 있습니다.

```
class Point
{
    int x, y;
public:
    // 방법 1. 초기화 리스트를 사용한 방법
    Point(int a, int b) : x(a), y(b)
    {
        // 방법 2. 생성자 { } 안에서 초기화 하는 방법
        x = a;
        y = b;
    }
};
```

그렇다면, 이 2 가지 방법의 차이점은 무엇일까요 ? 차이점을 이해 하려면 초기화와 대입의 차이점을 알아야 합니다.

- **초기화(initialize)와 대입(assignment)**

초기화는 객체를 만들 때 값을 넣는 것이고, 대입은 객체를 만든 후에 값을 넣는 것입니다.

```
int main()
{
    int n1 = 0; // 초기화. 객체를 만들면서 값을 넣고 있습니다.
```

```
    int n2;
    n2 = 0;     // 대입. 객체를 만든 후에 값을 넣고 있습니다.
}
```

[참고] 엄밀히 말하면 초기화는 "int n1(0)" 으로 표기 해야 합니다. 하지만, 여기서는 간단하게 설명하기 위해 "int n1 = 0" 으로 표기했습니다. 둘의 차이점은 "Advanced C++" 과정을 참고 하세요.

위 코드처럼 int 타입의 경우 초기화와 대입은 큰 차이가 없습니다. 하지만 n1, n2 가 int 타입이 아닌 사용자 정의 타입인 경우에는 n1 은 생성자 호출 한번으로 초기화 되지만, n2 는 만들 때 디폴트 생성자가 호출되고, 대입할 때 대입 연산자(3 장에서 배우게 됩니다.)를 호출하게 됩니다. 즉, 2 번의 함수 호출이 발생합니다.

또한, 상수일 경우는 초기화 될 수 있지만 대입될 수는 없습니다.

```
int main()
{
    Object n1(0);   // 생성자를 호출해서 초기화 합니다. 함수가 1번 호출됩니다.

    Object n2;      // 디폴트 생성자가 호출됩니다.
    n2 = 0;         // 대입연산자가 호출됩니다. 결국 함수가 2번 호출됩니다.

    const int c1 = 0; // ok. 상수는 초기화 될 수 있습니다.
    c1 = 0;           // error.상수는 대입될 수 없습니다.
}
```

결국, n1, n2 에 값을 넣는 것은 대입 보다 초기화를 하는 것이 보다 효율적인 코드가 됩니다.

- **생성자와 초기화**

초기화 리스트를 사용해서 멤버에 값을 넣은 것은 초기화 입니다. 하지만, 생성자의 블록 {} 안에서 값을 넣는 것은 대입 입니다.

```
class Point
{
    int x, y;
    const int c;
public:
```

```
    Point(int a, int b) : x(a),
                          y(b), // 초기화 입니다. y가 객체라면 생성자를 호출합니다.
                          c(b)  // ok.. 상수는 초기화 가능합니다.
    {
        x = a;
        y = b; // 대입입니다. y가 객체라면 대입연산자를 호출합니다.
        c = b; // error. 상수는 대입할 수 없습니다.
    }
};
```

일반적으로 C++에서는 멤버 data 를 초기화 할 때 생성자 블록 안에서 대입하는 것 보다는 초기화 리스트를 사용하는 방법을 권장합니다. 이 책 에서도 특별한 경우가 아니면 초기화 리스트를 사용해서 초기화 하도록 하겠습니다.

- **초기화 리스트가 반드시 필요한 경우**

다음과 같은 경우 반드시 초기화 리스트를 사용해서 초기화 해야 합니다.

① **클래스안에 상수 또는 참조 멤버가 있을 때**

② **클래스안에 디폴트 생성자가 없는 객체가 멤버로 있을 때**

```
class Point
{
public:
    Point(int a, int b) {}
};
class Test
{
    const int c;
    int& r;     // error. 참조는 초기화가 필요합니다.
    Point p1;   // error. 디폴트 생성자가 없습니다.
public:
    Test(int v)
    {
        c = 10; // error. 상수는 대입할수 없습니다.
    }
};
```

위의 경우 모두 초기화 리스트를 사용해서 초기화 하면 해결 할 수 있습니다.

```
class Test
{
    const int c;
    int& r;
    Point p1;
public:
    Test(int v) : c(v), r(v), p1(v, v)
    {
    }
};
```

C++11 문법을 적용하면 필드 초기화가 가능합니다. 하지만, 이경우는 생성자로 전달한 값이 아닌 상수를 사용해야 합니다.

```
class Test
{
    // 아래 처럼 초기화 할 수도 있습니다. 하지만, 이경우 생성자로 전달한 v를
    // 사용해서 초기화 할 수는 없습니다.
    const int c = 10;
    Point p1 = Point(0,0);
public:
    Test(int v)
    {
    }
};
```

- **주의 사항**

멤버 data 를 초기화 리스트에서 초기화 할 때 초기화 되는 순서는 리스트에 놓인 순서가 아니라 멤버가 놓은 순서로 초기화 됩니다.

```
class Point
{
public:
    int x;  // x가 먼저 선언 되어 있습니다.
```

```
    int y;

    Point() : y(0), x(y)    // y(0)이 먼저 있지만 x가 먼저 선언되어
    {                       // 있으므로 x(y)가 먼저 실행됩니다.
    }                       // 그런데, y 메모리는 아직 생성전이므로 undefined입니다.
};

int main()
{
    Point p;
    cout << p.x << endl; // undefined
    cout << p.y << endl; // 0
}
```

결국 초기화 리스트에서 멤버를 초기화 할 때 는 반드시 멤버가 놓인 순서로 초기화 해야 합니다.

11.2 위임 생성자(delegate constructor) - C++11

C++의 오래된 문법에서는 하나의 생성자에서 다른 생성자를 호출할 수 없었습니다. C++11 에 와서 비로서 하나의 생성자에서 다른 생성자를 호출 할 수 있게 되었습니다.

생성자에서 다른 생성자를 호출하는 방법은 초기화 리스트에서 생성자 호출 모양을 표시하면 됩니다.

```
class Point
{
private:
    int x;
    int y;
public:
    Point() : Point(0,0)  { // 다른 생성자 호출
    }
    Point(int a, int b) : x(a), y(b)  {
    }
};
```

주의 할 점은 아래와 같이 생성자 블록 안에서 다른 생성자를 호출하는 표현을 사용하는 것은 절대 다른 생성자를 호출하는 것이 아니고 임시객체를 생성하는 코드입니다. 사용하면 안됩니다.

```
class Point
{
    // ……
    Point()
    {
        Point(0,0);  // 다른 생성자 호출하는 표현이 아닙니다. 임시객체의 생성입니다.
    }
    Point(int a, int b) : x(a), y(b)  {
    }
};
```

[참고] 임시객체의 개념은 복사 생성자를 다룰 때 간단히 설명 됩니다. 보다 깊이 있는 설명은 "advanced C++" 과정을 참고 하시기 바랍니다.

- **코드 모양**

생성자를 선언과 구현으로 분리할 경우는 초기화 리스트는 아래 처럼 만들게 됩니다.

Point.h

```
class Point
{
    int x, y;
public:
    Point();
    Point(int a, int b);
};
```

Point.cpp

```
#include "Point.h"

Point::Point()
    : Point(0, 0)   // 다른 생성자 호출
{
}
Point::Point(int a, int b)
    : x(a), y(b)
{
}
```

- **멤버 데이터를 초기화 하는 방법 총정리**

결국, C++에서 멤버 data 를 초기화 하는 방법은 총 3 가지가 있습니다.

```
class Point
{
private:
    int x = 0;
    int y = 0; // 1. 필드 초기화(field initialize), C++11 부터 사용 가능한 방법
public:
    Point(int a, int b) : x(a), y(b) // 2. 초기화 리스트
    {
        // 3. 대입
        x = a;
        y = b;
    }
};
```

12장.복사 생성자(Copy Constructor)

12.1 복사 생성자 개념

모든 객체는 생성될 때 반드시 생성자가 호출 되어야 합니다. 적절한 모양의 생성자가 제공되지 않으면 객체를 생성할 수 없습니다. 아래 코드에서 main 함수를 생각해 봅시다.

```
class Point
{
public:
    int x, y;
    Point()            : x(0), y(0){} //1
    Point(int a,int b) : x(a), y(b){} //2
};
```

```
int main()
{
    Point p1;       // ok. 1번 생성자
    Point p2(1, 2); // ok. 2번 생성자
    Point p3(1);    // error
    Point p4(p2);   // 에러가 발생하지
}                   // 않습니다.
```

[참고] x, y 값을 화면 출력해서 확인해 보기 위해서 public 영역에 만들었습니다.

- **p1 의 경우는 인자를 전달하지 않으므로 인자가 없는 생성자(1 번) 이 호출됩니다.**
- **p2 는 인자가 int 타입 2 개를 가지는 경우 이므로 2 번 생성자를 호출해서 객체를 생성합니다.**
- **p3 의 경우 int 타입 하나를 인자로 가지는 생성자가 없으므로 compile error 입니다.**
- **p4 의 경우도 인자를 하나만 전달하고 있습니다. 그런데, 이경우는 컴파일을 해도 에러가 발생하지 않습니다. 왜 그럴까요 ?**

p4 객체는 생성할 때 인자로 p2 를 전달하고 있습니다. 그런데, p2 는 Point 타입이므로 결국 p4 가 생성되려면 Point 클래스 안에 자신의 타입 한 개를 인자로 가지는 생성자가 있어야 합니다.

```
Point p4(p2);// Point( Point ) 모양의 생성자가 필요합니다.
```

그런데, 사용자는 자신과 동일한 타입을 인자로 가지는 생성자로 만든 적이 없지만 컴파일 하면 에러가 발생하지 않습니다. 자신과 동일한 타입의 객체 하나를 인자로 받는 생성자를 복사 생성자라고 합니다. 모양은 다음과 같고, 사용자가 만들지 않으면 컴파일러가 제공해 줍니다.

```
// 사용자가 제공하지 않으면 컴파일러가 아래 모양의 복사 생성자를 제공합니다.
Point(const Point& p)
{
```

```
    x = p.x;
    y = p.y;
}
```

결국 복사 생성자는 객체가 자신과 동일한 타입의 객체로 초기화 될 때 사용됩니다. 이제부터 복사 생성자에 대해서 자세히 살펴 보도록 하겠습니다.

- **사용자가 만들지 않으면 컴파일러가 제공해 줍니다.**

복사 생성자는 디폴트 생성자와 유사하게 사용자가 만들지 않으면 컴파일러가 복사 생성자를 제공해 줍니다. 차이점은

✓ **디폴트 생성자는 어떠한 모양의 생성자도 제공하지 않을 때 컴파일러에 의해 생성됩니다.**

✓ **복사 생성자는 다른 형태의 생성자가 제공되어도 복사 생성자 자체가 없으면 컴파일러에 의해 생성됩니다.**

일반 생성자만 제공한 경우

```
class Point
{
public:
    Point() : x(0), y(0)  {  }
    // 일반 생성자는 있지만
    // 복사 생성자가 없으므로 컴파일러에
    // 의해 제공됩니다.
};

int main()
{
    Point p1;    // ok.
    Point p2(p1);// ok.
}
```

복사 생성자만 제공한 경우

```
class Point
{
public:
    Point(const Point& p) {  }
    // 복사 생성자도 생성자의 종류이므로
    // 컴파일러는 디폴트 생성자를
    // 하지 않습니다.
};
int main()
{
    Point p1;       // error
    Point p2(p1);   // ok.
}
```

물론, 둘다 제공하지 않으면 컴파일러에 의해 두개 모두 생성됩니다.

```
// 생성자와 복사 생성자를 모두 제공하지 않은 경우

class Point
{
public:
```

```
    // 사용자가 만든 생성자가 없으므로 컴파일러는 디폴트 생성자, 복사 생성자, 소멸자,
    // 대입연산자를 제공합니다.
};
int main()
{
    Point p1;    // ok.
    Point p2(p1);// ok.
}
```

이처럼 C++은 사용자가 만들지 않으면 컴파일러에 의해 자동 생성되는 요소들이 많이 있습니다. 뒷장에 가서 자세히 정리하도록 하겠습니다.

- **컴파일러가 제공하는 복사 생성자가 하는 일**

객체를 만들 때 다른 객체를 사용해서 초기화 했다면, 사용자는 2 개의 객체의 속성이 동일할 것이라고 기대 하게 됩니다.

```
int main()
{
    Point p1(1,2);  // ok.
    Point p2(p1);   // 대부분 p2도 1,2 로 초기화 될 것이라고 생각하게 됩니다.
}
```

그래서, 컴파일러가 제공하는 디폴트 생성자는 모든 멤버 data 를, 인자로 전달된 객체의 멤버 데이터 값으로 복사하는 코드를 제공합니다. Point 클래스의 경우 컴파일러에 의해 제공되는 복사 생성자의 모양은 다음과 같습니다. "복사 생성자"도 생성자의 일종이므로 초기화 리스트를 사용하는 것이 좋습니다.

```
// 컴파일러가 생성해 주는 복사 생성자. 모든 멤버를 복사해 주는 일을 합니다.
Point(const Point& p) : x(p.x), y(p.y)
{
}
```

물론, 사용자가 복사 생성자를 다시 만들어서 다른 구현을 제공해도 상관없습니다. 다음 코드를 작성하고 실행해 보세요

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

class Point
{
public:
    int x, y;
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    // 사용자가 직접 제공한 복사 생성자
    Point(const Point& p) : x(p.y), y(p.x)  // x, y를 바꿔서 대입했습니다.
    {
    }
};
int main()
{
    Point p1(1,2);  // ok.
    Point p2(p1);   // x, y를 바꾸어서 복사 했습니다.

    cout << p.x << endl; // 2
    cout << p.y << endl; // 1
}
```

[참고] 문법설명을 위한 예제일 뿐 입니다. 실전에서는 x, y 를 바꿔 복사할 경우는 없겠지요 ?

12.2 복사 생성자가 호출되는 경우

복사 생성자는 다음과 같은 경우에 사용됩니다.

① 객체를 만들 때 자신의 타입으로 초기화 되는 경우 호출됩니다.

② 함수 호출 시 인자를 call by value 로 전달할 때

③ 함수에서 객체를 값으로 리턴 할 때

테스트를 위해 Point 클래스를 아래 처럼 작성해 놓고 각각의 경우를 자세히 살펴 보도록 하겠습니다.

```
class Point
{
    int x, y;
public:
    Point()  { cout << "Point()" << endl; }
    ~Point() { cout << "~Point()" << endl; }
    Point(const Point& p)  { cout << "Point(const Point& p)" << endl; }
};
```

- **객체를 만들 때 자신의 타입으로 초기화 하는 경우**

C 에서 변수를 초기화 할 때 아래의 2 가지 방법으로 할 수 있습니다.

```
int a = 0; // = 로 초기화 하는 경우
int b(0);  // () 로 초기화 하는 경우
```

또한, C++11 의 "일관된 초기화(uniform initializer)" 문법을 사용하면 {}로 초기화도 가능합니다. 따라서, 아래의 코드에서 p2, p3, p4, p5 는 모두 복사 생성자를 사용해서 객체를 만드는 코드입니다.

```
int main()
{
    Point p1;           // 디폴트 생성자
    Point p2(p1);       // 복사 생성자
    Point p3 = p1;      // 복사 생성자
```

```
    Point p4{ p1 };     // 복사 생성자
    Point p5 = { p1 };  // 복사 생성자
}
```

[참고] () 를 사용한 초기화와 =를 사용한 초기화는 완전히 동일하지는 않습니다. 변환에 관련된 차이점이 있는데, 본 과정의 범위를 넘어서는 내용이므로 "Advanced C++ 과정"을 참고하시기 바랍니다.

- **함수 호출 시 인자를 call by value 로 전달할 때**

함수를 호출할 때 인자를 값으로 전달 받으면 복사 본이 생성 됩니다. 이때 복사 생성자가 사용됩니다.

```
// 결국, Point p2=p1 으로 전달받게 되므로  복사 생성자가 사용됩니다.
void foo(Point p2)
{
}
int main()
{
    Point p1;
    foo(p1);
}
```

위코드의 수행 결과는 다음과 같습니다.

```
Point()                 // p1객체 생성
Point(const Point& p)   // foo 함수의 인자인 p2 객체가 생성될때
~Point()                // foo 함수 종료시 p2 파괴
~Point()                // main 함수 종료시 p1 파괴.
```

결국 함수에 객체를 전달할 때 값으로 전달 받으면, 복사본이 생성되므로

- ✓ **복사본에 의한 추가 메모리 할당**
- ✓ **복사생성자, 소멸자 호출에 따른 성능저하가 있게 됩니다.**

따라서, 함수의 인자는 const 참조를 사용하는 것이 좋습니다. 다음 코드는 생성자와 소멸자를 각각 한번만 호출하는 코드 입니다.

```
void foo(const Point& p2) // p2는 p1을 가리키는 다른 이름 일 뿐 객체가 아닙니다.
{
}
int main()
{
    Point p1;
    foo(p1);
}
```

- **함수가 객체를 값으로 리턴 할 때.**

함수가 객체를 값으로 전달받으면 복사본이 생성되는 것처럼, 함수가 객체를 값으로 리턴 해도 복사본이 생성됩니다. 흔히, 이와 같은 복사본을 흔히 "임시객체" 라고 합니다. 이때, 리턴 용도의 임시객체를 만들 때도 복사 생성자가 사용됩니다.

```
Point p;        // 전역 객체

Point foo()
{
    return p;   // 이 순간 리턴용 임시객체가 생성됩니다.
                // p를 복사 해서 만들기 때문에 복사 생성자가 사용됩니다.
}
int main()
{
    foo();      // 리턴 값으로 돌아온 객체는 p가 아닌 리턴용 임시객체 입니다.
                // 리턴용 임시객체는 함수 호출문장 끝에서 파괴 됩니다.
}
```

[참고] 리턴용 임시객체는 함수 호출 구문의 끝에서 파괴됩니다.

위 코드에서는 분명히 코드 상으로는 객체가 전역객체 p 밖에 없습니다. 하지만, 실행해 보면 객체가 디폴트 생성자, 복사 생성자, 그리고 소멸자가 2 회 호출되는 것을 볼 수 있습니다. 결국 객체는 2 개가 생성되었다가 파괴 되고 있습니다.

```
Point()                 // 전역변수인 p객체 생성
Point(const Point& p)   // p를 복사 해서 만든 리턴용 임시객체 생성
~Point()                // foo()함수 호출이 종료되면 리턴용 임시객체 파괴
```

```
~Point()                // main 함수 종료후에 전역변수인 p객체 파괴
```

[참고] 이와 같은 임시객체의 개념은 C++을 처음 배우는 분들이 대부분 어렵게 느끼는 내용입니다. 보다 깊은 임시객체의 개념은 기본 서적에서 다루기는 어려우므로 "Advanced C++"을 참고 하시기 바랍니다.

이경우, 참조를 리턴 해서 임시객체가 생성되지 않도록 할 수 도 있습니다.

```
Point& foo() // 값이 아닌 참조를 리턴 합니다. 임시객체가 생성되지 않습니다.
{
    return p;
}
```

12.3 객체와 RVO (Return Value Optimization)

어떤 함수가 객체를 리턴 할 때

- ✓ **값 타입으로 리턴 하면 임시객체가 생성되어서 리턴 됩니다.**
- ✓ **참조 타입으로 리턴 하면 임시객체가 생성되지 않습니다.**

임시객체는 함수 호출식에서만 사용되고, 함수를 호출하는 문장이 끝나면 바로 파괴 됩니다. 바로 파괴되는 객체에 값을 넣는 것은 의미가 없습니다. 그래서 임시객체는 등호의 왼쪽편에 올 수 없습니다. 아래 코드를 참고 하세요.

```
Point p(1, 2);   // p는 전역 객체 입니다.

Point  foo() { return p; } // p를 복사한 임시객체가 리턴 됩니다.
Point& goo() { return p; } // p의 별명이 리턴 됩니다.

int main()
{
    // x멤버가 public 에 있을때
    foo().x = 10;   // error. 임시객체에 값을 넣을 수 없습니다.
    goo().x = 10;   // ok.    임시객체가 아니라 전역객체 p의 별명입니다.
                    //        p.x = 10 이 됩니다.
}
```

만약, 함수가 전역객체가 아닌 지역 객체를 리턴 한다면 어떨까요 ?

지역변수의 경우는 함수 호출 종료시 파괴 되므로 절대 참조를 리턴 하면 안됩니다. 미정의 동작(undefined) 발생합니다.

```
Point& foo()
{
    Point p;
    return p;   // 함수 호출 종료시 p는 파괴 됩니다.
                // 파괴된 객체의 참조를 리턴 할 수는 없습니다. undefined 입니다.
}
```

결국 값 타입으로 리턴 할 수밖에 없는데 이경우 임시객체가 생성되게 됩니다. 이때, 객체를 만든 후 리턴 하지 말고 만들면서 리턴 하면 훨씬 효율 적인 코드를 작성할 수 있습니다.

```
Point foo()
{
    Point p(1,2);   // 객체를 만들 때 생성자가 호출됩니다.
    return p;       // 리턴용 임시객체 때문에 복사 생성자가 호 출됩니다.
}
Point goo()
{
    return Point(1,2);  // 임시객체사용해서 리턴합니다.
}
```

C++에서 **"클래스이름()"** 를 사용하면 사용자가 임시객체를 생성할 수 있습니다. goo 함수 처럼 임시객체를 사용해서 리턴 하면 객체가 하나만 생성되므로 foo 함수 보다 효율적입니다.

이와 같이 임시객체를 사용해서 리턴 하는 기술을 "리턴 값 최적화(RVO, Return Value Optimization)" 라고 부릅니다. 요즘에서 컴파일러가 발전해서 foo 함수 처럼 만들어도 컴파일러의 최적화 과정을 거치면 goo 함수처럼 변경해 주는 경우가 많이 있습니다. (Named RVO, NRVO 라고 합니다.)

본 과정은 기본 과정이므로 임시객체에 대해서는 이정도로 소개만 하겠습니다. 보다 자세한 임시객체 내용은 "C++ Intermediate" 과정을 참고 하시기 바랍니다.

13장.객체 복사(Object Copy)

13.1 디폴트 복사 생성자와 얕은 복사, 깊은 복사

객체를 만들 때 생성자가 호출되고, 객체가 파괴 될 때 소멸자가 호출됩니다. 따라서, 생성자에서 자원을 할당한 경우 소멸자에서 자원을 반납하면 자원 관리를 자동으로 할 수 있습니다. 다음의 People 클래스를 생각해 봅시다.

```
class People
{
    char* name;
    int   age;
public:
    People(char* n, int a) : age(a)
    {
        name = new char[strlen(n)+1];   // 생성자에서 자원을 할당 하고 있습니다.
        strcpy(name, n);
    }
    ~People()
    {
        delete[] name;          // 생성자에서 할당된 자원은 소멸자에서 반납합니다.
    }
};
int main()
{
    People p1("kim", 2);
}
```

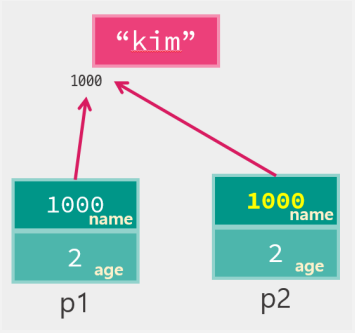
아무 문제 없어 보이는 이 코드는 main 함수를 다음과 같이 작성하면 컴파일 하면 문제가 없지만 실행시간에 에러가 나오게 됩니다. 왜 그럴까요 ?

```
int main()
{
    People p1("kim", 2);
    People p2(p1); // 실행시간에 에러가 발생합니다.
}
```

p2 의 경우 객체를 생성할 때 p1 을 인자로 사용하고 있으므로 복사 생성자가 사용됩니다. 그런데, 사용자가 복사 생성자를 제공하지 않았으므로 디폴트 복사 생성자가 사용됩니다.

다음 코드와 오른쪽의 그림을 보고 생각해 봅시다.

```
class People
{
    // ……
    // 컴파일러가 제공하는
    // 디폴트 복사 생성자의 모양입니다.
    People(const People& p)
        : name(p.name),
          age(p.age) {}
};
int main()
{
    People p1("kim", 2);
    People p2(p1);  // 이순간 디폴트 복사 생성자를
                    // 사용합니다.
}
```



디폴트 복사 생성자는 기본적으로 모든 멤버를 복사 하므로 포인터 멤버인 name 도 복사 합니다. 이때, p1 이 할당된 메모리의 주소를 p2 의 name 도 같이 사용하게 됩니다. 즉, p1 이 할당한 메모리 공간을 p2 객체도 같이 사용하게 됩니다.(오른쪽 그림 참고)

이때, p2 가 파괴 되면 소멸자에서 name 이 가리키던 메모리를 파괴 하게 됩니다. 그럼, 이제 p1 의 name 에 있는 주소는 이미 파괴된 메모리를 가리키게 됩니다.(dangling 포인터)

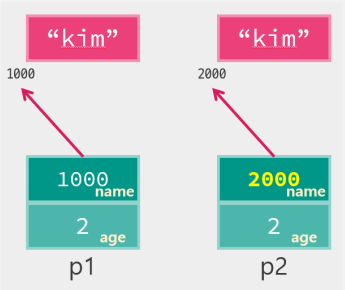
그리고, p1 이 파괴 될 때 p1 의 소멸자가 이미 파괴된 메모리를 다시 delete 하게 되는 문제가 발생합니다.

이처럼 클래스 멤버에 포인터가 있고, 생성자에서 메모리를 동적으로 할당할 때 디폴트 복사 생성자는 메모리 주소만 복사 하게 됩니다. 즉, 2 개의 객체가 하나의 메모리를 가리키게 됩니다. 이런 현상을 **"얕은 복사(shallow copy)"** 라고 합니다.

▪ 깊은 복사 (Deep Copy)

얕은 복사의 문제를 해결하려면 컴파일러가 제공하는 디폴트 복사 생성자를 사용하지 말고 사용자가 직접 복사 생성자를 제공해야 합니다. 이때, 포인터가 아닌 멤버는 아무 문제 없으므로 그냥 복사하면 됩니다. 하지만, 포인터의 경우 메모리 주소를 복사 하지 말고, 메모리를 새로 할당해서 메모리의 내용을 복사해야 합니다. 이처럼 메모리의 주소가 아닌 메모리 내용을 복사하는 것을 깊은 복사(Deep Copy)라고 합니다.

```
class People
{
public:
    // ……
    // 사용자가 만드는 복사 생성자입니다.
    People(const People& p)
        : age(p.age) // 포인터 멤버가 아닌 경우는
                     // 그냥 복사합니다.
    {
        // 포인터멤버는 메모리 할당후 메모리 자체를
        // 복사 합니다.
        name = new char[strlen(p.name) + 1];
        strcpy(name, p.name);
    }
};
int main()
{
    People p1("kim", 2);
    People p2(p1);  // 이제는 깊은 복사가 수행됩니다.
}
```



[참고] 복사 생성자를 깊은 복사로 만든 경우 대입연산자(assignment operator)도 깊은 복사로 구현해야 합니다. 연산자 재정의를 배울 때 다시 이야기 하도록 하겠습니다.

결국 핵심은,

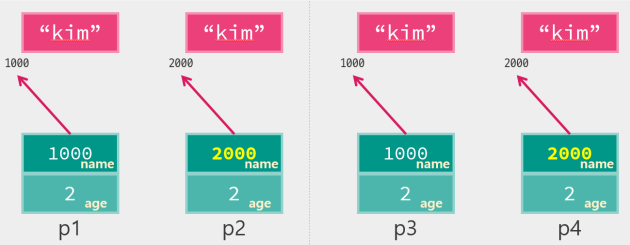
"클래스 내부에 포인터 멤버가 있고, 동적 할당된 메모리가 있다면 디폴트 복사 생성자는 얕은 복사(Shallow Copy) 현상을 일으킵니다. 반드시, 사용자가 복사 생성자를 제공해서 문제를 해결해야 합니다"

이와 같은 얕은 복사의 해결책은 깊은 복사만 있는 것은 아닙니다. 다음 절에서 다른 해결책도 살펴보도록 하겠습니다.

13.2 참조 계수를 사용한 복사 기술

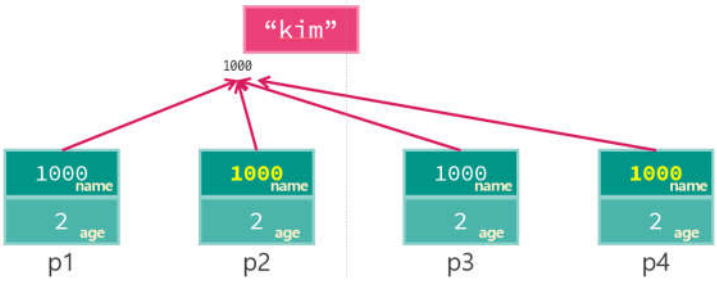
깊은 복사 방식은 동일한 내용을 가진 메모리가 여러 번 할당된다는 단점이 있습니다. 다음의 코드가 실행되었을 때 메모리 구조를 생각해 봅시다.

```
int main()
{
    People p1("kim", 2);
    People p2(p1);
    People p3(p1);
    People p4(p1);
}
```



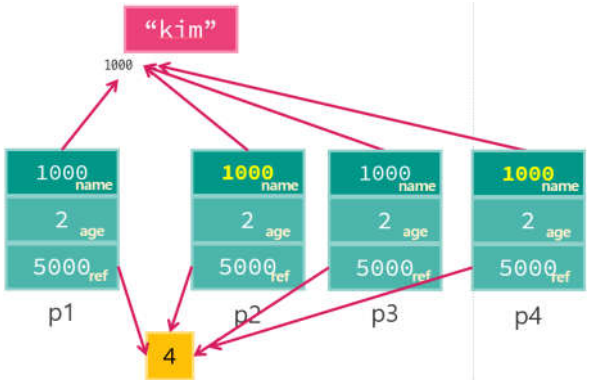
이코드는 "kim"이라는 동일한 문자열을 가진 메모리를 4 번이나 할당해야 합니다. 만약, 문자열의 길이가 긴 경우 메모리 낭비가 될 수 있습니다.

그렇다면, 어차피 동일한 문자열 이므로 아래 그림 처럼 되도록 하면 어떨까요 ?



얕은 복사일 경우의 그림입니다. 그런데, 이경우의 문제점은 p1~p4 의 객체 중 하나라도 파괴되면서 자원을 delete 하면 다른 객체들은 자원을 사용할 수 없다는 점입니다.

그렇 다면 아래 처럼 몇 명이 자원을 사용하는 개수를 관리하고, 소멸자에서는 자원을 바로 파괴 하지 말고 개수를 줄이다가 개수가 0 이 되었을 때 비로소 자원을 파괴 하면 어떨까요 ?



이런 방식으로 자원을 관리하는 것을 "참조계수(reference counting)"이라고 합니다. 완성된 코드는 다음과 같습니다. 코드에서 주의 깊게 살펴 볼 것은

① 생성자에서 자원(이름)뿐 아니라 참조계수를 관리하기 위해 int 를 할당하고 있는 점.

② 복사 생성자에서 얕은 복사후에 참조계수를 증가하고 있는 코드

③ 소멸자에서 참조계수를 줄인후, 0 이되는 경우 자원과 참조계수용 메모리를 delete 한다는 점입니다.

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

class People
{
    char* name;
    int   age;
    int*  ref;  // 참조계수를 관리할 메모리 주소
public:
    People(char* n, int a) : age(a)
    {
        name = new char[strlen(n)+1];
        strcpy(name, n);
        ref = new int;  // 참조계수용 변수를 할당하고
        *ref = 1;       // 1로 초기화 합니다.
    }

    // 복사 생성자는 얕은 복사후에 참조계수를 증가합니다.
    People(const People& p) : name(p.name), age(p.age), ref(p.ref)
    {
        ++(*ref);
    }
    ~People()
```

```
    {
        if (--(*ref) == 0) // 참조계수를 줄이고 0이 되면 메모리를 delete 합니다.
        {
            delete[] name;
            delete ref;
        }
    }
};
int main()
{
    People p1("kim", 2);
    People p2(p1);
    People p3(p1);
    People p4(p1);
}
```

[참고] 복사 생성자에서 "reference counting" 기술을 사용할 경우, 대입연산자(assignment operator) 도 재정의 해야 합니다. 연산자 재정의를 배울 때 다시 살펴 보도록 하겠습니다.

13.3 복사 금지를 사용한 해결책.

얕은 복사의 해결책 으로 깊은 복사, 참조계수 방법 외에 객체를 복사할 경우 컴파일 에러가 나오게 하는 복사 금지 기술도 있습니다.

```
int main()
{
    People p1("kim", 2);
    People p2(p1);  // 이순간 컴파일 에러가 나오게 합니다.
}
```

[참고] 디폴트 복사 생성자 사용시에는 "compile time error"가 아닌 "runtime error"가 발생하게 됩니다.

이 경우는 People 클래스를 사용할 때 "사람은 절대 복사 될 수 없다" 라는 의도를 가지고 설계하는 방식입니다. 그런데, 문제는 복사 생성자는 사용자가 만들지 않아도 컴파일러가 제공하므로 모든 타입의 객체는 항상 복사가 가능합니다.

객체를 복사 될 수 없게 하려면 컴파일러가 복사 생성자를 제공하지 못하게 해야 합니다. C++11 의 "delete function" 문법을 사용하면 됩니다.

```
class People
{
    // ……
    People(const People& p) = delete;   // 복사 생성자를 삭제 합니다. 컴파일러는
                                        // 더 이상 복사 생성자를 제공하지 않습니다.
};
int main()
{
    People p1("kim", 2);
    People p2(p1);  // 삭제된 함수를 호출하므로 "compile-time error" 입니다.
}
```

C++11 이 나오기 이전에는 (C++98/03) 에서는 복사를 금지하기 위해서 복사 생성자를 private 영역에 만드는 기법을 사용했습니다.

```
class People
{
    // ……
private:
```

```
    People(const People& p);// 복사 생성자를 private 영역에 선언만 합니다.
                            // 사용자가 복사 생성자의 선언을 제공했으므로
                            // 컴파일러는 복사 생성자를 제공하지 않습니다.
                            // 하지만, 구현이 없으므로 링커 에러 입니다.
                            // 결국, 객체는 복사 될 수 없습니다.
};
int main()
{
    People p1("kim", 2);
    People p2(p1);  // 삭제된 함수를 호출하므로 "compile-time error" 입니다.
}
```

[참고] 복사 생성자의 구현부를 제공하지 않은 이유는 멤버함수나 friend 함수안에서의 복사도 막기 위해서 입니다. 구현부가 있을 경우, private 영역이므로 외부 함수에서의 복사는 막을 수 있지만, 멤버함수나 friend 함수에서의 복사는 허용됩니다. 그래서, 선언만 제공하고 구현을 제공하지 않는 것입니다. 하지만, C++11 의 등장으로 이런 기법 보다는 "delete function" 문법을 사용하는 것이 좋습니다.

얕은 복사의 또 다른 해결책은 "소유권 이전"의 기술이 있습니다. 오래전 부터 사용되던 이 기술은 C++11 에서 "move 생성자" 라는 새로운 문법으로 발전하게 됩니다. 그런데, move 생성자를 이해하기 위해서는 임시객체, rvalue reference 등 어려운 내용이 많이 있기 때문에 기본 과정이 아닌 "C++ Intermediate" 과정에서 배우게 됩니다.

결국 얕은 복사의 해결책은

① **깊은 복사(deep copy)**

② **참조 계수(reference counting)**

③ **복사 금지**

④ **소유권 이전 - C++11 move constructor, "C++ Intermediate" 과정 참고**

등의 기법이 있습니다.

14장.정적 멤버(static member)

객체를 만들면 생성자가 호출되고 객체가 파괴 될 때 소멸자가 호출 됩니다. 그렇다면, 클래스 안에 cnt 멤버 data 를 만들고

- ✓ **생성자에서 ++cnt**
- ✓ **소멸자에서 --cnt**

를 수행하면 몇 개의 객체를 만들었는지 파악할 수 있지 않을 까요 ?

이 특징을 활용해서 현재 객체가 몇 개나 만들어져 있는지 개수를 파악하는 코드를 생각해 보겠습니다.

```
class Car
{
    int color;
public:
    int cnt = 0;

    Car()  { ++cnt; }
    ~Car() { --cnt; }
};
int main()
{
    Car c1, c2;
    cout << c2.cnt << endl; // ?
}
```

그런데, 문제는 클래스의 멤버 데이터인 cnt 는 객체당 하나씩 따로 만들어 지므로 cnt 는 2 가 아니라 각각 1 이 됩니다.

그렇다면 전역변수를 사용하면 어떨까요 ?

```
int cnt = 0;

class Car
{
    int color;
public:
    Car()  { ++cnt; }
```

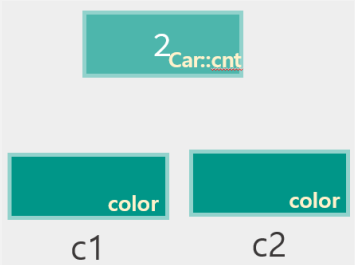
```
    ~Car() { --cnt; }
};
int main()
{
    Car c1, c2;
    cout << cnt << endl; // ok.. 이제 2가 나옵니다.
    cnt = 10;            // 그런데, 문제는 전역변수는 누구나 변경할 수 있습니다.
    cout << cnt << endl;
}
```

전역변수는 객체당 하나가 아니므로 모든 객체가 공유 할 수 있습니다. 그런데, 문제는 전역변수는 누구나 접근해서 값을 변경할 수 있습니다. 그렇다면, 전역변수처럼 모든 Car 의 객체가 공유 하는데, 외부에서 접근하지 못하게 할 수 없을 까요 ?

14.1 정적 멤버 데이터 (static member data)

static 멤버 data 를 사용하면 모든 객체가 공유 하는 멤버 data 를 만들 수 있습니다. Car 의 객체를 여러 개 만들어도 static 멤버 데이터는 모든 객체가 공유 하게 됩니다

```
class Car
{
    int color;
public:
    static int cnt;

    Car()  { ++cnt; }
    ~Car() { --cnt; }
};
int Car::cnt = 0;   // static 멤버 data는 반드시
                    // 클래스 외부에 선언해야 합니다.
int main()
{
    // Car 객체가 없어도 static 멤버 data는 메모리에
    // 존재 합니다. "클래스이름::정적멤버데이타이름"
    // 이름으로 접근합니다.
    cout << Car::cnt << endl; // 0

    Car c1, c2;
```



```
    // 객체 이름으로 접근할 수도 있습니다.
    cout << c1.cnt << endl; // 0
}
```

static 멤버 변수가 좀 복잡하므로 한 줄 씩 자세히 대해서 좀 자세히 살펴 보겠습니다.

- **클래스 외부에 선언이 필요합니다.**

static 멤버 데이터는 클래스 내부에 static 키워드를 사용해서 선언을 해야 하고, 클래스 외부에도 반드시 선언을 추가해야 합니다. 외부 선언시에는 static 키워드를 표시하지 않습니다.

```
class Car
{
public:
    static int cnt;
};
int Car::cnt = 0;   // 클래스 외부에 선언해야 합니다.
                    // 외부 선언시 static 키워드는 사용하지 않습니다.
```

- **객체를 만들지 않아도 메모리에 존재 합니다.**

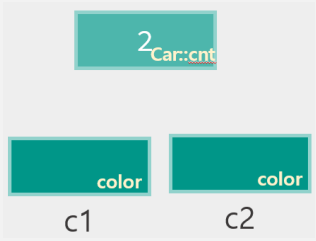
static 멤버 데이터의 수명은 전역변수와 동일합니다. Car 객체를 하나도 생성하지 않아도 메모리에 존재하며 프로그램이 종료 될 때 메모리에 놓이게 됩니다. 객체가 없어도 메모리에 존재하므로 "클래스이름::변수이름"으 로 접근 할 수 있습니다.

```
int main()
{
    // Car 객체가 없어도 static 멤버 data는 메모리에 존재 합니다.
    // "클래스이름::정적멤버데이타이름" 이름으로 접근 있습니다.
    cout << Car::cnt << endl; // 0
}
```

- **static 멤버 data 는 객체의 크기에 포함되지 않습니다.**

객체를 생성하면 일반 멤버 데이터는 객체당 하나씩 생성됩니다. 하지만, static 멤버 데이터는 전역변수와 유사하므로 프로그램 실행시 메모리에 놓이게 되고 각 객체를 만들 때는 추가로 생성되는 것은 없습니다.

```
int main()
{
    Car c1, c2;
    cout << sizeof(c1) << endl; // 4
}
```



- **static 멤버 데이터에 접근하는 2 가지 방법**

static 멤버 데이터에 접근 할 때는 아래의 2 가지 방법을 사용할 수 있습니다.

① **"클래스이름::멤버이름" 으로 접근**

② **"객체이름.멤버이름" 으로 접근**

위 두가지 방법 중에 에서 객체이름을 사용해서 접근하는 경우 해당 멤버가 static 멤버 인지, 일반 멤버 혼란 스러울수 있으므로 클래스이름을 사용해서 접근하는 것이 가독성 측면에서는 좋은 코드입니다.

```
int main()
{
    Car c1, c2;
    cout << Car::cnt << endl;   // "클래스이름::멤버이름" 으로 접근
                                // cnt가 static 멤버임을 쉽게 알 수 있습니다.
    cout << c1.cnt << endl;     // "객체이름.멤버이름"으로 접근
                                // cnt가 static 멤버 인지 일반 멤버인지
                                // 구별 할 수 없습니다.
}
```

[참고] java, C#에서는 static 멤버에 접근하려면 "클래스이름.멤버이름"으로 접근합니다. 객체이름으로는 접근할 수 없습니다

14.2정적 멤버 함수 (static member function)

static 멤버 데이터는 전역변수와 유사하지만 멤버 데이터이므로 접근지정자를 사용해서 외부의 잘못된 사용을 막을 수 있습니다. 객체의 개수는 생성자/소멸자 에서만 자동으로 관리 하는 것이 좋습니다.

```
class Car
{
    int color;
    static int cnt;
public:
    Car()  { ++cnt; }
    ~Car() { --cnt; }
    int getCount() { return cnt;}
};
int Car::cnt = 0;

int main()
{
    Car c1, c2;
    //Car::cnt = 10; // error. private 영역에 있으므로 외부에서는 접근할수 없습니다.
    cout << c1.getCount() << endl; // 갯수를 꺼내고 싶을때는 멤버 함수를 사용합니다.
}
```

그런데, 아직 한가지 문제가 남아 있습니다. 정적 멤버인 cnt 는 Car 객체를 만들지 않아도 메모리에 존재 하지만 getCount 함수를 호출하려면 반드시 객체를 만들어야 합니다. 즉, Car 를 만들지 않고는 cnt 값을 확인할 방법이 없습니다.

- static member function

static member function 은 객체 없이도 호출할 수 있는 멤버 함수 입니다. getCount 함수를 static 멤버 함수로 만들면 Car 객체를 만들지 않아도 cnt 값을 꺼낼수 있습니다.

```
class Car
{
    static int cnt;
    // ……
    static int getCount() { return cnt;}
};
```

```
int Car::cnt = 0;

int main()
{
    //static 멤버 함수는 "클래스이름::함수이름" 으로 호출 할 수 있습니다.
    cout << Car::getCount() << endl;

    Car c1, c2;
    cout << c1.getCount() << endl; // "객체이름.함수이름()" 으로도 호출할 수 있습니다.
}
```

정적 멤버 데이터와 마찬가지로 정적 멤버 함수를 호출 할 때는 "객체이름.함수이름()"으로 호출할 경우 함수가 일반 멤버 함수 인지 정적 멤버 함수인지 구별 할 수 없습니다. 되도록 이면 **"클래스이름::함수이름()"** 을 사용하는 것이 좋습니다.

정리 하면

정적 멤버 데이터 (static member data)

모든 객체가 공유하는 멤버 데이터 로서, 객체를 생성하지 않아도 메모리에 존재합니다. 전역변수와 유사하지만 접근지정자를 사용할 수 있고, 클래스 안에 포함되므로 관련 있는 데이터를 타입 안에 묶어서 관리 할 수 있게 됩니다.

정적 멤버 함수 (static member function)

객체 없이 호출 될 수 있는 멤버 함수 입니다. 결국, 객체가 없어도 호출 되므로 일반 함수와 유사하지만 접근 지정자를 활용 할 수 있고, 특정 타입과 관련된 함수를 묶어서 관리 할 수 있다는 특징이 있습니다.

14.3정적 멤버 관련 주의 사항

정적 멤버 관련해서 몇가지 주의 사항을 살펴 보도록 하겠습니다

- **정적 멤버 함수에서는 정적 멤버만 접근 할 수 있습니다.**

정적 멤버 함수와 정적 멤버 데이터의 특징을 잘 생각하고, 아래 코드를 생각해 봅시다.

```
class Test
{
    int data;
    static int cnt;
public:
    static void foo()
    {
        data = 0; // A
        cnt  = 0; // B
    }
};
int Test::cnt = 0;

int main()
{
    // static 멤버 함수는 객체없이
    // 호출 될수 있습니다.
    Test::foo();
}
```

cnt 는 정적 멤버 데이터 이므로 객체가 없어도 메모리에 존재 합니다. 하지만 data 는 일반 멤버 이므로 객체를 생성할 때 메모리에 만들어 지게 됩니다. 즉, 객체가 없는 경우는 메모리에는 cnt 만 존재 하게 됩니다.

그런데, 정적 멤버 함수는 객체 없이 호출 할 수 있습니다. 그렇다면 foo 가 호출되었을 때 메모리에는 data 는 없고, cnt 만 있게 됩니다. 결국, 정적 멤버 함수 안에 서는 정적 멤버 데이터만 접근 할 수 있습니다.

```
static void foo()
{
    data = 0; // A error. 정적 멤버 함수 에서는 일반 멤버에 접근할 수 없습니다.
```

```
    cnt  = 0; // B ok.    정적 멤버 함수 에서는 정적 멤버는 접근할 수 있습니다.
}
```

- **static 멤버 변수와 초기화**

클래스 멤버 데이터 초기화 관련해서 아래의 규칙이 있습니다.

① **C++11 부터는 멤버 data 를 만들 때 클래스 안에서 바로 초기화 할 수 있습니다.**

② **정적 멤버 데이터의 초기값은 반드시 클래스 외부 선언에서 해야 합니다.**

③ **static const 멤버 데이터는 클래스 외부선언이 없어도 됩니다. 또한, 클래스 안에서 바로 초기화 할 수 있습니다.**

```
class Test
{
    int data1 = 0;          // ok. C++11 부터 허용되는 문법.
    static int data2 = 0;   // error. 초기값은 반드시 외부 선언에서 해야 합니다.

    static const int data3 = 0; // ok. 외부 선언이 없어도 됩니다.
                                //     내부 초기화를 허용합니다.
};
int Test::data2 = 0;
```

- **선언과 구현의 분리**

클래스를 선언과 구현으로 분리할 때 코드 작성 규칙은 다음과 같습니다.

① **정적 멤버 함수의 static 키워드는 선언부에만 표기해야 합니다. 구현부에는 static 을 표기하지 않습니다.**

② **static 멤버 데이터의 경우 클래스의 외부 선언은 구현파일에 작성해야 합니다.**

다음 코드는 이번 장에서 배운 Car 클래스의 완전한 소스 코드입니다. 반드시 실습해서 실행해 보시기 바랍니다.

Car.h　　Car.cpp

```
class Car
{
    int color;
    static int cnt;
public:
    Car();
    ~Car();

    static int getCount();
};
```

```
#include "Car.h"

// static 멤버 data의 외부 선언
int Car::cnt = 0;

Car::Car()  { ++cnt;}
Car::~Car() { --cnt; }

int Car::getCount() {
    return cnt;
}
```

main.cpp

```
#include <iostream>
#include "Car.h"
using namespace std;

int main()
{
    Car c1, c2;
    cout << Car::getCount() << endl;
}
```

g++로 빌드 하려면 아래 처럼 하면 됩니다.

```
g++ main.cpp car.cpp -std=c++11
```

15장.상수 멤버 함수(const memer function)

15.1 상수 멤버 함수의 개념

멤버 함수를 만들 때 함수의 () 뒤에 const 를 붙이는 함수를 상수 멤버 함수 라고 합니다. 상수 멤버 함수 안에서는 모든 멤버를 상수 취급 하게 됩니다.

```
class Point
{
    int x, y;
public:
    // ……
    void print() const // 상수 멤버 함수
    {
        x = 10; // error. 상수 멤버함수에서는 모든 멤버를 상수 취급합니다.

        cout << x << ", " << y << endl; // 멤버의 값을 읽는 것은 문제 없습니다.
    }
};
```

상수 멤버 함수는 어렵지는 않은 문법입니다. 그런데, 이 문법에서 중요한 것은 "왜 상수 멤버 함수가 필요한가?" 입니다. 지금부터 상수 함수의 필요성에 대해서 살펴 보도록 하겠습니다.

15.2 상수 함수의 필요성 - 상수 객체는 상수 멤버 함수만 호출할 수 있다.

다음의 Point 클래스와 main 함수를 잘 생각해 보세요. 설명을 위해 멤버 데이터 x, y 를 public 영역에 놓았습니다. 또한, print 멤버 함수 포함 모든 멤버 함수는 상수 함수가 아닙니다.

```
class Point
{
public:
    int x, y;
    Point(int a = 0, int b = 0) : x(a), y(b) {}
    void set(int a, int b) { x = a;  y = b;  }
    void print() {  cout << x << ", " << y << endl; }
};
int main()
{
    const Point p(1, 1); // 상수 객체
    p.x = 10;       // x는 public 영역에 있지만 상수 객체이므로 error입니다..
    p.set(10, 10);  // 될까요 ?
    p.print();      // 될까요 ?
}
```

비록, 멤버 data 가 x,y 가 public 영역에 있지만 객체 p 는 상수 객체이므로 아래 코드는 에러입니다.

```
p.x = 10;       // x는 public 영역에 있지만 상수 객체이므로 error입니다.
```

그렇다면, 멤버 데이터를 접근하는 코드가 아닌 멤버 함수를 호출하는 아래 코드는 어떻게 될까요 ?

```
p.set(10, 10);  // ?
```

이 경우도 set 을 호출하고 나면 p 의 상태가 변경되므로 역시 error 입니다.

그렇다면 print 멤버함수는 내부적으로 멤버 데이터의 값을 변경하지 않고 있습니다. 호출 할 수 있을까요 ?

```
p.print();
```

보통 클래스를 만들 때 클래스의 선언과 멤버 함수의 구현을 각각 헤더 파일과 소스 파일로 분리한 후에 만들게 됩니다. 또한, 사용자는 헤더 파일만 포함하면 해당 클래스를 사용할 수 있습니다. 이때, 과연 컴파일러가 print 함수의 선언만 보고 멤버 데이터의 값을 변경하는지 하지 않는지 알 수 있을까요 ? 알 수 없습니다. 그래서 컴파일러는 상수 객체인 p 를 사용해서는 대해서는 print 함수도 호출할 수 없습니다.

Point.h

```
class Point
{
public:
    int x, y;
    Point(int a = 0, int b = 0);
    void set(int a, int b);

    // 선언만을 보고 멤버 함수가 멤버의
    // 값을 변경하는지는 알 수 없습니다.
    void print();
};
```

main.cpp

```
#include "Point.h"

int main()
{
    const Point p(1, 1); // 상수 객체
    p.x = 10;       // error
    p.set(10, 10);  // error
    p.print();      // error
}
```

그런데, 아무리 상수 객체라 하더라도 상태를 변경할 수는 없지만 값을 출력할 수는 있어야 하지 않을까요 ? 상수객체에 대해 print 멤버 함수를 사용할수 있게 하려면 컴파일러에게 print 함수에서는 멤버의 값을 변경하지 않겠다면 알려주어야 합니다. 이때 필요한 것이 바로 상수 멤버 함수입니다.

위 예제에서 print 멤버함수를 상수 멤버 함수로 변경하고 실행해 보면 아무 문제 없이 잘 컴파일 됩니다.

결국 핵심은

"상수 객체는 상수 멤버 함수만 호출 할 수 있다."

입니다.

- **상수 객체 (const object)**

C++ 에서는 상수 객체를 많이 사용합니다. 간단한 예제 하나를 더 살펴 보겠습니다. 다음 코드는 아주 간단한 Rect 클래스 입니다.

```
class Rect
```

```
{
    int x, y, w, h;
public:
    Rect(int a = 0, int b = 0, int c = 0, int d = 0)
        : x(a), y(b), w(c), h(d) {}
    int getArea() { return w * h; }
};
// call by value 보다는 const & 를 사용해야 합니다 - 레퍼런스 참고
void foo(const Rect& r)
{
    int n = r.getArea(); // 면적을 구할수 있을까요 ?
}
int main()
{
    Rect r(1, 1, 10, 10);   // 상수 객체가 아니므로
    int n = r.getArea();    // getArea()를 호출하는 것은 문제가 없습니다.
    foo(r);
}
```

main 함수에서 만든 Rect 객체 r 은 상수 객체가 아니므로 getArea()를 호출하는 것은 문제가 없습니다.

그런데, 문제는 foo 함수입니다. 객체를 전달 받을 때 는 call by value 는 동일한 크기의 객체를 하나 더 만들게 되므로 성능이 좋지 않습니다. 그래서 call by value 대신에 const 레퍼런스를 사용하는 것이 좋습니다. 그런데, 이때 참조 r 은 상수 참조 입니다. 상수 객체는 상수 함수만 호출 할 수 있으므로 error 입니다.

```
void foo(const Rect& r)
{
    int n = r.getArea(); // error.
}
```

아무리 상태를 변경 할 수 없는 고정된 사각형이라 하더라도, 면적을 구할 수는 있어야 하지 않을까요 ? getArea()함수는 반드시 상수 함수로 만들어야 합니다.

```
int getArea() const { return w * h; }
```

일반적으로 클래스의 멤버 함수를 만들 때 객체의 상태를 변경하지 않은 모든 멤버 함수 반드시 상수 멤버 함수로 만들어야 합니다.

[참고] 소위 말하는 getter(get 으로 시작하는 멤버 함수들)은 대부분 상수 멤버함수로 만들어야 합니다.

15.3상수 멤버 함수 관련 주의 사항

마지막으로 상수 멤버 함수 관련 몇가지 주의할 점을 살펴 보도록 하겠습니다.

▪ 상수 함수 안에서는 모든 멤버 함수는 상수 취급 된다

상수 멤버 함수안에서는 모든 멤버는 상수 입니다. 아래 코드를 생각해 보세요..

```
class Point
{
    int x, y;
public:
    int* get_address_of_x() const   // error. 상수 멤버 함수 이므로 x는 const 입니다.
    {                               // 리턴 타입은 const int* 가 되어야 합니다.
        return &x;
    }
};
```

[참고] private 멤버 데이터의 주소를 리턴 하는 멤버 함수는 좋은 코드는 아닙니다. 단지 문법 설명을 위한 코드입니다.

▪ 동일 이름의 상수 함수와 비 상수 함수

동일한 이름의 상수함수와 비 상수 함수를 동시에 제공할 수 있습니다. 상수 객체의 경우 상수 멤버 함수를 비 상수 객체는 비 상수 함수를 호출하게 됩니다. 아래 코드는 꼭 컴파일해서 실행해 보세요.

```
class Test
{
public:
    void foo()       { cout << "foo()"       << endl; } // 1
    void foo() const { cout << "foo() const" << endl; } // 2
};
int main()
{
    Test t1;
    t1.foo();   // 1번이 호출 됩니다.  1번이 없으면 2번을 호출합니다.

    const Test t2;
    t2.foo();   // 2번이 호출 됩니다. 2번이 없으면 error입니다.
}
```

[참고] 함수 오버로딩이 아닙니다. 함수 오버로딩은 인자가 달라야 합니다. 위의 경우는 인자도 완전히 동일합니다.

- **선언과 구현의 분리**

상수 멤버 함수를 선언과 구현으로 분리할 경우 선언부와 구현부 모두에 const 를 표기해야 합니다.

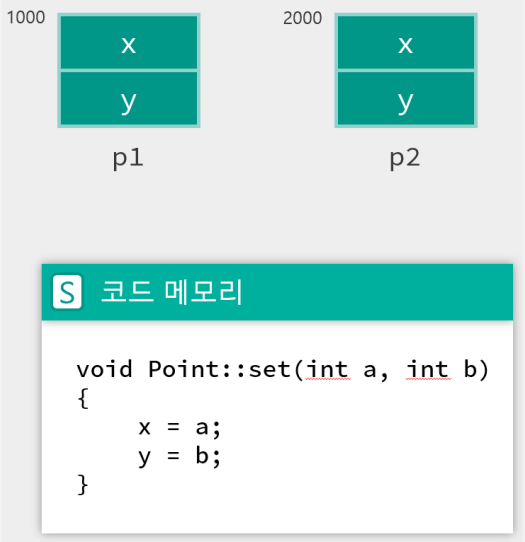
```
class Test
{
public:
    void foo() const;
    void goo() const;
};
void Test::foo() {          // error. const는 선언과 구현 모두에 있어야 합니다.
}
void Test::goo() const {    // ok..
}
```

16장. this 포인터

16.1 this 개념

객체를 2 개 만들면 정적 멤버가 아닌 일반 멤버 데이터는 메모리에 각각 2 개씩 놓이게 됩니다. 하지만 멤버 함수는 객체를 여러 개 만들어도 코드 메모리에 한 개만 놓이게 됩니다. 다음에서 코드를 생각해 봅시다.

```
class Point
{
    int x;
    int y;
public:
    void set(int a, int b)
    {
        x = a;
        y = b;
    }
};
int main()
{
    Point p1;
    Point p2;
    p1.set(10, 20);
}
```



그렇다면, 이때 생각해 볼 것은 메모리에는 분명히 x 와 y 가 2 개 있습니다. 그렇다면 하나밖에 없는 set()함수에서는 자신이 사용하는 x 가 p1 의 x 인지, p2 의 x 인지를 어떻게 알 수 있을까요 ?

C++ 컴파일러는 멤버함수를 컴파일 할 때 하면 아래 처럼 객체의 주소가 멤버 함수의 인자로 전달되는 코드를 생성하게 됩니다. 왼쪽 코드의 원리를 표현한 것이 오른쪽 코드 입니다.

사용자가 만든 코드

```
class Point
{
    int x;
    int y;
public:
    void set(int a, int b)
    {
```

컴파일러가 생성한 코드

```
class Point
{
    int x;
    int y;
public:
    void set(Point* const this, int a, int b)
    {
```

```
        x = a;
        y = b;
    }
};
int main()
{
    Point p1;
    Point p2;

    p1.set(10, 20);
}
```

```
        this->x = a; // this는 p1의 주소 입니다.
        this->y = b; // 결국 p1의 x,y에 값을 넣습니다.
    }
};
int main()
{
    Point p1;
    Point p2;

    set(&p1,10, 20);//p1의 주소를 set함수로 전달합니다.
}
```

[참고] 일반적으로 함수의 인자는 스택으로 전달되고, 객체의 주소는 보통 레지스터로 전달되게 됩니다. 그래서, 실제 함수 인자와 객체 주소를 전달하는 방식은 약간 차이가 있지만 개념 설명을 위해 객체의 주소가 함수의 인자 처럼 전달되는 것으로 표현 했습니다. 더욱 자세한 내용은 "Advanced C++"에서 다루게 됩니다.

결국 멤버 함수안에서 this 는 멤버 함수를 호출할 때 사용한 객체의 주소가 됩니다. p1.set(10,20)으로 호출된 경우는 결국 p1 의 주소인 1000 번지에 있는 x 에 10 을 넣게 됩니다.

- **this 사용하기**

이와 같은 this 는 사용자가 직접 사용해도 됩니다. this 를 사용하면 자신을 호출 할 때 사용한 객체의 주소를 알 수 있습니다. 아래 코드를 실행해 보세요

```
class Point
{
    int x;
    int y;
public:
    void foo() { cout << this << endl;  }
};
int main()
{
    Point p1;
    Point p2;

    // 아래 2줄은 동일한 주소가 출력됩니다.
    cout << &p1 << endl;
    p1.foo();

    // 아래 2줄 역시 동일한 주소가 출력됩니다.
```

```
    cout << &p2 << endl;
    p2.foo();
}
```

[참고] 위 코드에서 foo 함수는 단순히 문법설명을 위해 만들었습니다. 어떤 의미가 있는 함수는 아닙니다.

16.2 this 활용

객체의 주소를 나타내는 this 는 다양한 경우에 활용 될 수 있습니다.

- **이름 충돌시 멤버 변수에 접근하기 위해**

멤버 변수의 이름과 함수 인자의 이름이 동일 할 때 멤버에 접근하려면 this 를 사용합니다.

```
class Point
{
    int x;
    int y;
public:
    void set(int x, int y)
    {
        // 여기서는 x는 인자의 x 를 가리킵니다.
        // 멤버 변수를 가르키려면 this->x 를 사용합니다.
        this->x = x;
        this->y = y;
    }
};
```

- **this 를 리턴 하는 멤버 함수**

멤버 함수가 this 를 리턴 하면 멤버 함수의 함수 호출을 연속적으로 수행 할 수 있습니다.

```
class Car
{
    int color;
public:
    Car* Go1() { return this; }
    Car& Go2() { return *this; }  // 주의!, 반드시 참조를 리턴해야 합니다.
                                  // 값을 리턴하면 임시객체가 생성됩니다.
};
int main()
{
    Car c;
```

```
    // 함수를 연속적으로 호출 할 수 있습니다.
    c.Go1()->Go1()->Go1();
    c.Go2().Go2().Go2();
}
```

[참고] cout 의 원리가 이렇게 되어 있습니다. 연산자 재정의 를 배울 때 cout 을 직접 만들어 보도록 하겠습니다.

Go2 함수의 경우 주의 할 점은 반드시 참조를 리턴 해야 한다는 점입니다. 값을 리턴 하면 임시객체가 생성되게 됩니다.

이 외에도 this 는 다양하게 활용될 수 있습니다. 다양한 C++ 코드를 경험하시면 this 를 사용하는 많은 기술을 보실 수 있습니다.

SECTION 3

Inheritance

이번 장에서는 상속(Inheritance) 문법에 대해서 살펴 보도록 하겠습니다.

지금까지의 클래스 문법은 단지, 하나의 타입을 만드는 문법이었지만, 상속은 서로 연관된 타입을 설계해서 어떻게 객체지향 프로그램을 구성하는지를 배우는 과정입니다.

상속 문법은 객체 지향 프로그래밍을 이해하는 핵심 요소 입니다.

이번 장에서 다음과 같은 개념을 배우게 됩니다.

"상속의 개념, 가상함수와 다형성, 도형편집기 예제, 추상 클래스와 인터페이스, 함수 바인딩과 가상함수의 원리, RTTI"

17장. 상속(inheritance) 개념

17.1 상속의 기본개념

학사 관련 프로그램을 만드는 데, Student 와 Professor 클래스가 필요하다고 생각해 봅시다.

```
class Student
{
    string name;
    int    age;
    int    id;
};
```

```
class Professor
{
    string name;
    int    age;
    int    major;
};
```

[참고] 개념 설명을 위해 생성자, 멤버함수 등을 제외한 최대한 간결한 코드로 작성했습니다.

그런데, Student 와 Professor 은 모두 이름(name)과 나이(age) 라는 공통의 멤버를 가지고 있습니다. 이때는 위처럼 각각 만들지 말고 상속 문법을 사용하면 훨씬 간단하게 클래스를 만들 수 있습니다.

먼저, Student 와 Professor 는 사람이라는 공통의 특징이 있으므로 사람의 특징(이름, 나이)를 가진 클래스를 설계합니다.

```
class People
{
    string name;
    int    age;
};
```

그리고, Student 와 Professor 클래스를 만들 때 People 부터 상속 받으면 name, age 멤버를 물려 받을 수 있게 됩니다. Student 와 Professor 는 사람으로 서의 공통의 특징(name, age)을 제외한 자신만의 특성만 추가하면 됩니다.

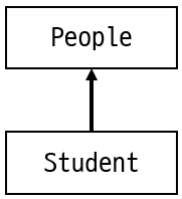
```
class Student : public People   // 상속 문법. People의 모든 멤버를 물려 받게 됩니다.
{
    int     id;
};
class Professor: public People
{
    int     major;
```

```
};
```

Student 클래스를 만들 때 People 클래스로 부터 상속을 받으려면 다음처럼 만들면 됩니다.

```
class Student : public People
```

이때, People 클래스를 **"Base Class"** 또는 **"Super Class"** 라고 부르고, Student 클래스를 **"Derived Class"** 또는 **"Sub Class"**라고 부릅니다. 이 책에서는 **"Base(기반) 클래스"**와 **"Derived(파생)클래스"** 라는 용어를 사용하도록 하겠습니다. 상속의 관계를 그림으로 표현할 때는 아래 처럼 표현합니다.



상속을 사용하면 멤버 데이터 뿐 아니라 멤버 함수 도 모두 물려 받게 됩니다. 그럼 이제 부터, 상속 관련된 다양한 문법에 대해서 살펴 보도록 하겠습니다.

17.2 상속과 접근 지정자

클래스가 가지고 있는 private 멤버는 자신의 멤버 함수에서는 접근할 수 있지만 클래스 외부에서는 접근 할 수 없습니다. 그렇다면, 파생클래스의 멤버함수에서는 접근할 수 있을까요 ?

기반 클래스에 있는 private 멤버는 파생클래스의 멤버 함수에서도 접근할 수 없습니다. 이때, protected 라는 접근 지정자를 사용하면 외부에서는 접근할 수 없지만 파생 클래스에서는 접근 가능한 멤버를 만들 수 있습니다.

```
class Base
{
private:
    int data1; // 자신의 멤버 함수와 friend 함수만 접근할 수 있습니다.
protected:
    int data2; // 자신의 멤버 함수와 friend 함수
               // 파상클래스의 멤버 함수와 friend 함수 에서 접근 할 수 있습니다.
               // 하지만, 클래스 외부에서는 접근할 수 없습니다.
public:
    int data3; // 클래스 외부에서도 접근 가능합니다.
};
class Derived : public Base
{
public:
    void foo()
    {
        data1 = 0; // error.
        data2 = 0; // ok.
    }
};
```

결국, 접근지정자를 총정리 하면 C++에는 3 개의 접근 지정자가 있습니다.

private	자신의 멤버 함수와 friend 함수에서만 접근 할 수 있습니다.
protected	자신의 멤버 함수와 friend 함수 파생 클래스의 멤버 함수와 파생클래스의 friend 함수에서 접근 할 수 있습니다.
public	모든 함수에서 접근 할 수 있습니다.

C#, java, swift 등 다른 객체 지향 언어에는 internal, package 등의 접근 지정자도 제공 되지만 C++은 위의 3 가지만 제공합니다.

17.3 상속과 생성자 소멸자

클래스가 기반 클래스를 가지고 있다면 객체 생성시 기반 클래스의 생성자가 먼저 호출되고 자신의 생성자가 호출됩니다. 소멸자는 반대로 자신의 소멸자가 먼저 호출되고, 기반 클래스의 소멸자가 마지막으로 호출됩니다.

아래 코드를 실행해 보면 기반 클래스의 생성자가 먼저 호출되고 소멸자는 반대로 기반 클래스의 소멸자가 마지막에 호출 되는 것을 볼 수 있습니다.

```
class Base
{
public:
    Base()  { cout << "Base()" << endl; }
    ~Base() { cout << "~Base()" << endl; }
};
class Derived : public Base
{
public:
    Derived()  { cout << "Derived()" << endl; }
    ~Derived() { cout << "~Derived()" << endl; }
};
int main()
{
    Derived d;
}
```

```
Base()
Derived()
~Derived()
~Base()
```

결국 파생 클래스를 만들면 기반 클래스의 생성자가 먼저 호출되게 되는데, 여기에는 숨겨진 원리가 있습니다. 지금부터 정확한 원리를 살펴 보도록 하겠습니다.

- **상속에서의 생성자 소멸자 호출의 정확한 원리**

상속을 사용하는 클래스에서 생성자, 소멸자 호출의 정확한 원리를 아는 것은 아주 중요합니다. 사용자가 어떤 클래스를 만들 때 상속을 사용하면 컴파일러는 컴파일시에 생성자 안에 기반 클래스의 생성자를 호출하는 코드는 넣어주고, 소멸자 에도 기반 클래스의 소멸자를 호출하는 코드를 넣어 주게 됩니다.

아래 코드를 잘 생각해 보세요.

사용자가 만든 코드

```
class Base
{
public:
    Base()      {  }
    Base(int a) {  }
    ~Base()     {  }
};
class Derived : public Base
{
public:
    Derived()      { }
    Derived(int a) { }
    ~Derived()     { }
};
int main()
{
    Derived d(1);
}
```

컴파일러가 생성해 주는 코드

```
class Derived : public Base
{
public:
    // 기반 클래스의 생성자를 호출하는
    // 코드를 넣어 줍니다.
    Derived()      : Base() { }
    Derived(int a) : Base() { }
    ~Derived()
    {
        // ……
        ~Base();
    }
};
int main()
{
    // 결국 아래 코드는 Derived의 생성자를
    // 호출합니다.
    Derived d(1);
}
```

결국 "Derived d(1)" 의 경우는 자신의 생성자만 호출하게 되는데, 생성자의 앞부분 (초기화 리스트) 에 컴파일러가 생성해준 기반 클래스의 생성자를 호출하는 코드가 있기 대문에 기반 클래스의 생성자를 먼저 실행하고, 돌아와서 자신의 생성자를 실행하게 되는 원리 입니다.

소멸자의 경우 자신의 구현을 먼저 실행하고 마지막 으로 기반 클래스의 소멸자를 호출하는 코드가 있기 때문에 기반 클래스 소멸자가 나중에 호출되는 것 처럼 보이게 됩니다.

또 하나, 확실하게 알아 두어야 하는 점은 인자가 한 개 있는 생성자의 경우도 기반 클래스의 생성자는 디폴트 생성자를 호출하도록 만들어진 다는 점입니다.

```
Derived(int a) : Base() { } // 컴파일러가 생성해준 코드는 기반 클래스의 디폴트 생성자
                            // 를 호출하게 됩니다.
```

따라서, 위 코드의 각 함수에 함수 자신의 이름을 출력하도록 코드를 추가하고 실행해 보면 아래의 결과를 볼 수 있습니다.

```
Base()          // 기반 클래스는 디폴트 생성자가 호출됩니다.
```

```
Derived(int)
~Derived()
~Base()
```

이 경우 기반 클래스를 디폴트 생성자가 아닌 다른 생성자가 호출되게 하려면 아래 처럼 사용자가 직접 기반 클래스의 생성자를 호출하면 됩니다.

```
Derived(int a) : Base(a) { } // 사용자가 직접 기반 클래스의 생성자 호출합니다.
```

- **기반 클래스에 디폴트 생성자가 없을 때**

기반 클래스에 디폴트 생성자가 없을 경우 반드시 사용자가 직접 기반 클래스의 생성자를 호출하는 코드를 작성해야 합니다.

```
class Base
{
    int id;
public:
    Base(int n) : id(n) {}
};
class Derived : public Base
{
public:
    Derived() {}    // error. 컴파일러가 생성해준 코드는 "Base()" 이므로 Base
                    // 디폴트 생성자를 호출하게 됩니다.
};
int main()
{
    Derived d;
}
```

이 경우 해결책은 사용자가 직접 기반 클래스를 생성자를 호출하는 코드를 제공해야 합니다.

```
class Base
{
    int id;
public:
```

```
    Base(int n) : id(n) {}
};
class Derived : public Base
{
public:
    Derived()      : Base(0) {}
    Derived(int n) : Base(n) {}
};
int main()
{
    Derived d;
}
```

17.4 protected 생성자

클래스를 만들 때 생성자를 protected 에 놓는 경우가 있습니다. protected 생성자는 어떤 의미를 가질 까요 ? 아래 코드를 잘 생각해 봅시다.

```
class Animal
{
protected:
    Animal() { }
};
class Dog : public Animal
{
public:
    Dog() {}
};
int main()
{
    Animal a; // error. 외부에서 protected 생성자를 호출할 수 없습니다.
    Dog    d; // ok.    Dog 생성자를 먼저 호출하고, Dog 생성자 안에서 Animal 생성자를
              //        호출합니다. 파생 클래스에서는 기반 클래스의 protected 멤버에
              //        접근할 수 있습니다.
}
```

결국, protected 생성자는

"자신 타입의 객체는 생성할 수 없지만 파생 클래스 타입의 객체는 생성 할 수 있다"

는 설계 의도를 담은 코드 입니다.

이것은 현실 세계와 매우 유사한데, 현실 세계에서 "동물(Animal)"은 객체가 존재 할 수 없는 추상적인 개념 이지만 "개(Dog)" 는 객체가 존재 할 수 있습니다.

18장.가상함수

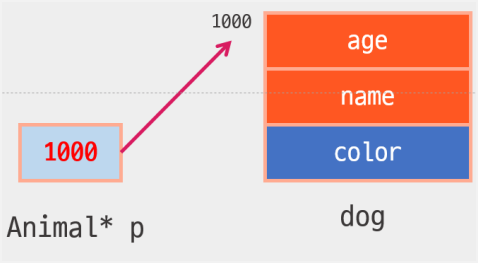
18.1 upcasting

일반적으로 C/C++ 언어에서는 포인터 변수에 다른 타입의 주소를 담을 수는 없습니다.

```
double d = 3.4;
int*   p = &d; // error. double 의 주소를 int* 에 담을 수는 없습니다.
```

하지만, 기반 클래스의 포인터에는 파생 클래스의 주소를 담을 수 있습니다. 기반 클래스의 포인터에 파생클래스의 주소를 담는 것을 upcasting 이라고 합니다.

```
class Animal
{
public:
    int    age;
    string name;
};
class Dog : public Animal
{
public:
    int color;
};
int main()
{
    Dog dog;
    Animal* p = &dog; // ok. Upcasting
}
```



위코드에서 Dog는 Animal 의 파생 클래스 이므로 Animal의 모든 멤버를 물려 받게 됩니다. 따라서, Dog 의 객체를 생성하면 메모리 앞쪽에는 Animal 의 모양이 있게 됩니다. 그러므로 Animal* p 로 Dog 객체를 가리 키는 것은 당연 합니다. 오른쪽 그림을 참고 하세요

18.2 Down Casting

Animal* p 로 Dog 를 가르 킬 때, p 를 사용해서는 Animal 의 멤버만 접근 할 수 있습니다. 비록 메모리에 Dog 의 고유한 멤버인 color 가 존재 하지만 p 를 사용해서 접근할 수는 없습니다. 만일 color 에 접근하고 싶다면 Animal* 형태인 p 를 다시 Dog* 로 캐스팅후 접근해야 합니다.

```
int main()
{
    Dog dog;
    Animal* p = &dog;   // Upcasting, 암시적 변환 가능합니다.

    p->age = 10;        // ok
    p->color = 0;       // error. Animal* 로는 Dog 고유의 멤버는 접근할 수 없습니다.

    // Dog* pDog = p;   // error. Downcasting은 암시적 캐스팅이 안됩니다.
    Dog* pDog = static_cast<Dog*>(p); // Downcasting
    pDog->color = 0;    // ok
}
```

[참고] Downcasting 을 위해서는 dynamic_cast 를 사용하는 것이 좋습니다. 뒤에 RTTI 를 다룰 때 설명됩니다.

기반 클래스의 포인터를 다시 파생 클래스 포인터 타입으로 캐스팅할 때는 반드시 명시적으로 캐스팅 해야만 합니다. 이런 캐스팅을 "Down Casting" 이라고 합니다.

결국 기반 클래스 타입으로 파생 클래스를 가르 킬 수는 있지만 기반 클래스로부터 상속 받은 멤버만 사용 할 수 있고, 파생 클래스의 고유한 멤버를 접근할 수는 없습니다. 접근 하려면 다시 파생 클래스 타입으로 캐스팅 해야 합니다. 이 특징을 어떻게 왜 활용하는지는 뒤에 예제에서 다루게 됩니다.

18.3 Upcasting 활용

upcasting 을 활용하면 동종(동일한 기반 클래스를 사용하는 타입)을 처리하는 함수나 동종을 보관하는 컨테이너를 만들 수 있습니다.

- **동종을 처리하는 함수**

어떤 동물의 나이가 10 살 이상인지를 조사하는 is_bigger_than_10() 이르는 함수를 생각해 봅시다.

```
bool is_bigger_than_10(Dog* p)
{
    return p->age > 10;
}
```

위 함수의 문제는 인자로 Dog* 를 사용하므로 이 함수에는 Dog 만 전달 할 수 있습니다. 다른 동물 Cat, Tiger 등을 처리 하려면 별도로 만들어야 합니다. 그런데, age 멤버는 기반 클래스인 Animal 클래스에 있으므로 Animal 로부터 파생된 모든 클래스에는 공통으로 존재 하게 됩니다. 그렇다면 인자로 Dog 를 사용하지 말고 Animal 을 사용하면 모든 동물(Animal 로부터 파생된 모든 클래스)을 처리하는 함수를 만들 수 있습니다.

```
bool is_bigger_than_10(Animal* p)
{
    return p->age > 10;
}
```

물론, is_bigger_than_10() 함수에는 모든 Animal 파생 클래스를 전달할 수 있지만 실제 사용시에는 Animal 의 멤버만 사용해야 합니다.

- **동종을 보관하는 컨테이너**

기반 클래스 포인터를 사용하면 동종을 보관하는 컨테이너를 만들 수도 있습니다

```
vector<Dog*>    v1; // Dog만 보관할수 있습니다.
vector<Animal*> v2; // 모든 동물(Animal의 파생클래스)를 보관합니다.
```

윈도우 OS 의 탐색기를 객체지향 프로그램으로 만든다고 생각해 봅시다. 폴더안에는 다양한 파일을 보관 할 수 있으므로 Folder 안에는 vector 나 list 등의 컨테이너가 있어야 합니다.

```
class Folder
{
    vector<File*> v;
};
```

Folder 안에는 File 뿐 아니라 Folder 자체도 보관할 수 있습니다. 그런데, vector<File*> 처럼 만들면 파일밖에 보관할 수 없습니다. Folder 가 File 뿐 아니라 Folder 도 보관하게 하려면 어떻게 해야 할까요 ? File 과 Folder 는 동일한 기반 클래스를 가져야 합니다.

```
// 파일과 폴더의 공통의 기반 클래스
class Item
{
    // ......
};
class File : public Item
{
};
class Folder : public Item
{
    vector<Item*> v;
};
```

Folder 가 File 뿐 아니라 Folder 자체도 보관 할 수 있다면 File 과 Folder 는 동일 기반 클래스를 가져야 한다는 이런 디자인 기술을 "Composite 패턴" 이라고 합니다. 이런 객체 지향 디자인 기술에 대한 자세한 이야기는 "디자인 패턴" 과정을 수강하시면 됩니다.

본 교재에서도 뒷장에 "도형편집기 만들기 예제"에서도 좀더 다루게 됩니다.

18.4 함수 오버라이드 (function override)

기반 클래스의 멤버함수를 파생 클래스가 재정의(override) 함수 있습니다. "function override" 라고 합니다. 이경우 Animal 의 객체를 만들면 Animal 의 멤버함수를, Dog 의 객체를 만들면 Dog 의 멤버 함수를 호출합니다.

```
class Animal
{
public:
    void Cry() { cout << "Animal Cry" << endl; }
};
class Dog : public Animal
{
public:
    // function override : 기반 클래스 함수를 파생 클래스에서 다시 만듭니다.
    void Cry() { cout << "Dog Cry" << endl; }
};
int main()
{
    Animal a;
    a.Cry(); // "Animal Cry"
    Dog d;
    d.Cry(); // "Dog Cry"
}
```

[참고] 함수 오버로딩(overloading)이 아닙니다. overloading 은 인자의 개수나 타입이 다른 경우 입니다. override 는 인자의 개수와 타입이 모두 같은 경우 입니다.

그런데, 만약 Upcasting 을 사용하면, 즉, Animal 포인터를 사용해서 Dog 를 가르킬 때, Animal 포인터를 사용해서 Cry 를 호출하면 Animal 의 Cry 가 호출될까요 ? Dog 의 Cry 가 호출될까요 ?

```
Animal* p = new Dog;
p->Cry();  // "Animal Cry" 를 호출할까요 ? "Dog Cry"를 호출 할까요 ?
```

이경우, 포인터 타입인 Animal 의 Cry 를 호출합니다. 실제 객체인 Dog 의 Cry 가 호출되게 하려면 Cry 함수를 가상함수로 만들어야 합니다.

18.5 가상 함수 (virtual function)

멤버 함수를 만들 때 함수 앞에 virtual 키워드를 붙이는 함수를 가상 함수 라고 합니다. 기반 클래스의 포인터로 파생 클래스를 가르킬 때, 파생 클래스가 재정의(override) 한 함수가 호출되게 하려면 가상함수로 만들어야 합니다. 아래 코드에서 Cry()가 일반 멤버 함수 일 경우(왼쪽)과 가상함수 일 경우(오른쪽) 각각 실행해 보면 결과가 달라 지게 됩니다.

```
class Animal
{
public:
    void Cry()
    {
        cout << "Animal Cry" << endl;
    }
};
class Dog : public Animal
{
public:
    void Cry()
    {
        cout << "Dog Cry" << endl;
    }
};
int main()
{
    Animal* p = new Dog;
    p->Cry(); // "Animal Cry"
    delete p;
}
```

```
class Animal
{
public:
    virtual void Cry()
    {
        cout << "Animal Cry" << endl;
    }
};
class Dog : public Animal
{
public:
    virtual void Cry()
    {
        cout << "Dog Cry" << endl;
    }
};
int main()
{
    Animal* p = new Dog;
    p->Cry(); // "Dog Cry"
    delete p;
}
```

일반적으로 포인터가 기반 클래스 라도 객체가 Dog 라면 Dog 의 멤버함수를 호출하는 것이 논리적으로 맞습니다.

그래서,

"파생클래스에서 재정의 하게 되는 함수는 가상함수로 만드는 것이 원칙입니다."

사실 문법적인 내용보다는 가상함수를 언제, 어떻게 활용하는지를 아는 것이 훨씬 중요합니다. 그런데, 먼저 가상함수의 문법적 특징을 정리하고 활용성은 예제를 통해서 좀더 살펴 보도록 하겠습니다.

- **파생 클래스에서 가상함수 재정의시 virtual 키워드는 붙여도 되고 안 붙여도 됩니다.**

가상함수를 재정의 할 때 virtual 키워드를 생략해도 됩니다. 하지만 이경우 가독성이 떨어 질 수 있으므로 되도록이면 붙이는 것이 좋습니다.

```
class Dog : public Animal
{
public:
    // 가상함수 재정의시 virtual 을 생략할 수 있습니다.
    // 기반 클래스에서 가상함수 만들었으면 재정의시 virtual 이 없어도 가상함수 입니다.
    void Cry()
    {
        cout << "Dog Cry" << endl;
    }
};
```

[참고] C++언어의 단점 중 하나가, "~ 해도 되고 안해도 되는데, 되도록이면 ~ 하는게 좋다." 라는 문법이 너무 많다는 점입니다.

- **멤버 함수를 선언과 구현을 분리 할 때는 선언에만 virtual 키워드를 표시합니다.**

```
class Animal
{
public:
    virtual void Cry();
};
// 구현시에는 virtual을 적으면 안됩니다.
void Animal::Cry()
{
    cout << "Animal Cry" << endl;
}
```

참고로, 상수함수, 정적 멤버 함수, 가상함수를 각각 선언과 구현으로 분리 할 때의 모양을 정리하면 다음과 같습니다.

```
class Test
{
public:
            void f1() const;// 함수 구현부에도 const를 붙여야 합니다.
    static  void f2();      // 함수 구현부에는 static을 붙이지 않습니다.
```

```
    virtual void f3();      // 함수 구현부에는 virtual을 붙이지 않습니다.
};
```

- **가상함수 재정의시 override 키워드를 사용하는 것이 좋습니다. - C++11**

가상함수를 재정의 할 때 함수 이름이나, 인자를 잘못 만드는 실수를 할 수도 있습니다. 하지만, 이경우 문제점은 에러가 나오지 않고 파생 클래스가 추가한 다른 함수로 인식된다는 점입니다. 이때는 override 키워드를 사용하면 컴파일 시에 에러가 나오게 할 수 있습니다.

```
class Base
{
public:
    virtual void f1() {}
    virtual void f2() {}
};
class Derived : public Base
{
public:
    // f1을 재정의 하려고 했는데 실수로 이름을  잘못적었습니다.
    // 에러가 나지 않고 파생 클래스가 추가한 새로운 함수로 인식됩니다.
    virtual void ff1() {}

    // 기반 클래스에는 ff2가 없습니다.
    virtual void ff2() override {} // error
};
```

override 키워드는 기반 클래스가 가진 가상함수를 재정의 하고 있다고 컴파일러에게 알려 주는 것입니다. 따라서, 동일한 이름의 가상함수가 기반 클래스에 없으면 에러가 발생합니다.

그렇다면, 차라리 모든 가상함수를 재정의 할 때 override 를 꼭 붙이라고 규칙을 정했으면 좋지 않을까요 ? 하지만, override 라는 문법은 C++98/03 까지는 없던 문법 입니다. C++11 에서 추가된 문법입니다. 따라서, 과거에 만들었던 코드를 계속 지원하기 위해 override 는 붙여도 되고 붙이지 않아도 됩니다. 정리하면

"가상함수를 재정의 할 때 override 를 붙여도 되고 붙이지 않아도 됩니다. 하지만, 가능하면 붙이는 것이 좋은 코드입니다."

- **final 키워드 - C++11**

C++11 에 추가된 final 키워드를 사용하면 파생 클래스가 가상함수를 재정의 할 수 없도록 만들 수 있습니다.

```
class Base
{
public:
    virtual void f1() {}
    virtual void f2() {}
};
class Derived1 : public Base
{
public:
    virtual void f1() override {}
    virtual void f2() override final{}// 이 가상함수는 Derived1 까지만 재정의 됩니다.
                                     // 이후의 파생 클래스에서는 재정의 할 수 없습니다.
};
class Derived2 : public Derived1
{
public:
    virtual void f1() override {}   // ok
    virtual void f2() override {}   // error. f2는 Derived1 에서 재정의 할 때 final을
                                    // 사용했기 때문에 더 이상 재정의 할 수 없습니다.
};
```

그럼, 도대체 final 같은 문법은 언제 사용 할까요 ? C++의 모든 문법이 왜? 언제? 사용되는지를 알려면 수 많은 예제를 통해서 객체 지향 프로그래밍의 다양한 기법을 경험해야 비로소 문법의 의미를 정확히 파악할 수 있습니다. 일단은, 정확한 문법을 이해하고, 많은 예제와 코드를 경험하는 것이 최선의 방법입니다.

18.6 가상 소멸자

객체를 만들면 생성자가 호출되고 객체가 파괴되면 소멸자가 호출됩니다. 생성자와 소멸자는 항상 쌍으로 호출되므로 보통 생성자에서 자원을 획득하면 소멸자에서 자원을 반납 하게 됩니다.

```
class Test
{
public:
    Test()  { cout << "자원획득" << endl; }
    ~Test() { cout << "자원반납" << endl; }
};
int main()
{
    Test t;
}
```

그런데, 상속을 사용하는 클래스에서는 주의할 점이 있습니다. 아래 코드를 생각해 보세요.

```
class Base
{
public:
    Base()  { }
    ~Base() { }
};
class Derived : public Base
{
public:
    Derived()  { cout << "자원획득" << endl; }
    ~Derived() { cout << "자원반납" << endl; }
};
```

생성자에서 자원을 획득하고 소멸자에서 자원을 반납하는 아주 기본 적인 코드 입니다. 그런데, 이제 Derived 클래스를 다음 처럼 사용한다고 생각해 봅시다.

```
int main()
{
    Base* p = new Derived; // 이순간 Derived 의 생성자가 호출되고 자원이 획득됩니다
    delete p;   // 자원 반납이 될까요 ?
```

```
}
```

먼저 Derived 의 객체를 생성했으므로 Derived 생성자가 호출되고 자원이 획득됩니다. 그런데, 아래처럼 객체를 파괴 할 때,

```
delete p;   // p의 타입은 Derived가 아닌 Base* 입니다.
```

p 가 가르키는 메모리가 해지되고 소멸자가 호출되어야 합니다. 그런데, p 의 타입이 Derived 아니고 Base 이므로 컴파일러는 이순간 Base 가 파괴 된다고 생각하고 소멸자는 Base 의 소멸자만 호출되게 됩니다. 즉, Derived 의 소멸자가 호출되지 않게 됩니다.

결국, 기반클래스의 포인터로 파생 클래스를 가르 킬 때, delete 로 객체를 파괴하면 기반 클래스의 생성자만 호출된다는 것입니다.

결국 자원 반납은 되지 않으므로 버그가 발생합니다. 해결책은 기반 클래스의 소멸자를 가상함수로 만들어야 합니다.

```
class Base
{
public:
    Base()  { }
     virtual ~Base() {} // 기반 클래스의 소멸자는 반드시 가상함수 이어야 합니다.
                        // 기반 클래스의 소멸자가 가상함수이면, 모든 파생 클래스의
                        // 소멸자는 자동으로 가상 소멸자가 됩니다.
};
class Derived : public Base
{
public:
    Derived()  { cout << "자원획득" << endl; }
    ~Derived() { cout << "자원반납" << endl; }
};

int main()
{
    Base* p = new Derived;
```

```
    delete p;   // 이제 이순간, Derived의 소멸자가 호출됩니다.
}
```

결론은, 어떤 클래스를 만들 때 상속 문법을 사용하게 된다면 기반 클래스의 소멸자는 반드시 가상함수로 만들어야 합니다.

19 장. 예제로 배우는 객체지향 프로그래밍

19.1 도형 편집기 예제

파워 포인트 같은 프로그램을 보면 다양한 도형을 편집 할 수 있습니다. 이와 같은 프로그램을 만들어 가면서 객체지향 프로그래밍의 원리를 살펴 보겠습니다.

중요한 내용이 많으므로 각 단계별로 만들어 보겠습니다.

Step1. 각 도형을 타입화 한다.

도형 편집기 같은 프로그램서는 "사각형, 원, 삼각형"등 의 다양한 도형을 다루게 됩니다. 사각형을 표현하기 위해 int 타입 4 개를 사용하는 것 보다는 Rect 라는 타입이 있으면 편리하므로 각 도형을 타입으로 만들어 보겠습니다. 만들어 과정에서는 설명에 꼭 필요한 요소만 만들고, 멤버 데이터/생성자/소멸자 등은 생략하도록 하겠습니다.

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

class Rect
{
public:
    void Draw() { cout << "Rect Draw" << endl; }
};
class Circle
{
public:
    void Draw() { cout << "Circle Draw" << endl; }
};
int main()
{
    vector<Rect*>   v1; // Rect 만   저장할 수 있습니다.
    vector<Circle*> v2; // Circle 만 저장할 수 있습니다.
}
```

위 코드는 main 함수에서 만들어진 도형을 보관하기 위해 vector 를 사용하는데, 문제는 vector<Rect*> 는 Rect 만 보관할수 있고, vector<Circle*>은 Circle 만 보관 할 수 있다는 점입니다.

도형을 따로 보관 할 경우, 어떤 도형을 먼저 만들었는지 등을 관리하기가 어렵습니다. 모든 종류의 도형을 하나의 컨테이너에 보관 할 수 있다면 좋지 않을까요 ?

Step2. 기반 클래스 "Shape" 의 도입

각 도형의 동일한 기반 클래스를 도입하면 하나의 컨테이너에 모든 도형을 보관할 수 있습니다. 이전의 코드에 Shape 클래스를 도입하고, Rect 와 Circle 클래스의 기반 클래스로 사용합니다. 그리고, main 함수에서 vector<Shape*> 를 보관하면 모든 도형을 보관 할 수 있게 됩니다.

```
class Shape
{
};
class Rect   : public Shape { // …… };
class Circle : public Shape { // …… };
int main()
{
    vector<Shape*>   v;
    // v에는 종류에 상관없이 모든 도형을 보관할 수 있습니다.
    v.push_back(new Rect);
    v.push_back(new Circle);
}
```

보통 상속의 장점은

"기존 클래스의 특징을 물려받아 새로운 특징을 추가 할 수 있다는 재사용 성 입니다."

또한, 위처럼

"연관된 클래스들을 묶어서 관리할 수 있다"

는 특징도 있습니다.

이제 사용자에게 입력을 받아서 다양한 도형을 만들수 있도록 코드를 추가해 보겠습니다. 사용자가 1 을 입력 하면 Rect 을 만들고, 2 를 입력하면 Circle 을 만들고, 9 을 입력하면 지금까지 만든 모든 도형의 Draw 함수를 호출하도록 만들겠습니다. main 함수를 다음 처럼 변경합니다.

```
int main()
{
```

```
    vector<Shape*>   v;

    int cmd;
    while (1)
    {
        cin >> cmd;
        if      (cmd == 1) v.push_back(new Rect);
        else if (cmd == 2) v.push_back(new Circle);
        else if (cmd == 9)
        {
            for (auto p : v)
                p->Draw();    // compile time error
        }
    }
}
```

어렵지 않은 코드인데, 문제는 아래 코드에서 compile error 가 발생합니다.

```
for (auto p : v)
    p->Draw();    // compile time error
```

왜 에러일까요 ?

Step3. 파생클래스의 공통의 특징은 기반 클래스에 있어야 한다.

기반 클래스의 포인터로 파생 클래스를 주소를 담을 수 있지만 파생클래스의 고유한 멤버에 접근할 수는 없습니다.

```
class Shape { };
class Rect : public Shape
{
public:
    void Draw() {  }
};
int main()
{
    Shape* p = new Rect;
    p->Draw();  // error. 기반 클래스가 파생클래스를 가리킬 때
                // 파생 클래스의 고유한 멤버에 접근할 수는 없습니다.
```

```
}
```

이전의 코드에서는 사용자의 입력에 따라 Rect 또는 Circle 을 만들지만 모든 도형을 같은 컨테이너에 보관하기 위해 vector 안에는 Shape*를 보관했습니다. 따라서, vector 에서 값을 하나 꺼냈을 때 얻게 되는 포인터는 Shape* 입니다. 그런데 Shape* 안에는 Draw 함수가 없으므로 에러가 나오게 됩니다.

```
int main()
{
    vector<Shape*>   v;

    // ……
    for (auto p : v)
        p->Draw();  // p는 실제로는 Rect 혹은 Circle을 가리키지만
                    // p의 타입은 Shape*입니다
}
```

해결책은 다음의 2 가지중에 하나입니다.

① p 를 Rect 또는 Circle 로 캐스팅해서 사용하는 방법

② Shape 안에 Draw 를 만드는 방법.

그런데, 어떤 도형을 만들었는지는 실행시간에 사용자 입력에 따라 달라지므로 코드를 작성하는 시점에서는 Rect 인지 Circle 인지 알 수 없습니다. 결국 해결책은 Shape 클래스 안에도 Draw 함수를 제공해야 합니다.

여기서 핵심은

"모든 도형의 공통적인 특징은 반드시 기반 클래스인 Shape 에도 있어야 합니다."

그래야만, 기반 클래스의 Shape* 로 도형을 관리할 때 해당 특징을 사용할 수 있게 됩니다.

[참고] 실제로, Shape* 타입의 p 가 실제로 어떤 도형을 가르 키는 지를 조사할 수 있습니다. RTTI 기술을 사용하면 가능합니다. 책의 뒤장에서 다루게 됩니다.

Step4. 파생 클래스가 override 하게 되는 함수는 반드시 가상함수로 만들어야 합니다.

결국 모든 도형(Rect, Circle)에 Draw 함수가 있다면 Shape 함수에도 반드시 Draw 함수를 제공해야 합니다. 즉, Shape 클래스를 설계 할 때부터 모든 도형의 공통의 특징이 무엇인가를 생각해서 Shape 안에 멤버로 제공해야 합니다.

```
class Shape
{
public:
    void Draw() { cout << "Shape Draw" << endl; }
};
class Rect : public Shape
{
public:
    void Draw() { cout << "Rect Draw" << endl; }
};
int main()
{
    Shape* p = new Rect;
    p->Draw();  // ok. Shape 안에도 Draw 함수가 있으므로 error가 발생하지 않습니다.
                // 하지만, Rect의 Draw가 아닌 Shape의 Draw가 호출됩니다.
}
```

이제는, Shape*타입의 포인터 p 를 사용해서 Draw 를 호출 할 수 있습니다. 그런데, 마지막 문제는 실제 객체는 Rect 이지만 포인터 p 가 Shape* 타입이므로 Shape 의 Draw 를 호출하게 됩니다. 해결책은 Draw 함수를 가상함수로 만들어야 합니다.

일반적으로 기반 클래스를 설계할 때

"파생 클래스에서 override 하게 되는 멤버함수는 반드시 가상함수로 만들어야 한다."

는 규칙이 있습니다.

Step5. 핵심 정리

지금 까지의 과정을 정리하면 다음과 같습니다.

① 모든 도형을 타입으로 만들면 편리하기 때문에 Rect, Circle 클래스를 만들었습니다.

② Rect, Circle 을 하나의 컨테이너에 보관하기 위해 기반 클래스인 Shape 를 만들었습니다.

③ 모든 도형의 공통의 특징은 기반 클래스인 Shape 안에 있어야 Shape* 를 사용해서 모든 도형의 특징을 사용할 수 있습니다. 그래서, Shape 안에 Draw 가 있어야 합니다.

④ Draw 함수는 파생 클래스에서 override 하게 되므로 기반 클래스인 Shape*로 사용할 때 파생 클래스의 Draw 를 호출되게 하려면 반드시 가상함수로 만들어야 합니다.

다음은 완성된 코드 입니다.

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

class Shape
{
public:
    virtual void Draw() { cout << "Shape Draw" << endl; }
    virtual ~Shape() {} // 뒷장의 "가상 소멸자" 항목 참고
};
class Rect : public Shape
{
public:
    virtual void Draw() { cout << "Rect Draw" << endl; }
};
class Circle : public Shape
{
public:
    virtual void Draw() { cout << "Circle Draw" << endl; }
};

int main()
{
    vector<Shape*>   v;

    int cmd;
    while (1)
    {
```

```
        cin >> cmd;
        if      (cmd == 1) v.push_back(new Rect);
        else if (cmd == 2) v.push_back(new Circle);
        else if (cmd == 9)
        {
            for (auto p : v)
                p->Draw();
        }
    }
}
```

19.2 다형성 (Polymorphism) 과 개방 폐쇄의 법칙 (Open Close Principle)

객체 지향 프로그래밍의 3 가지 특징을 흔히 "캡슐화, 상속성, 다형성" 이라고 합니다. 각각의 특징을 간단히 요약하면

- **캡슐화 : 멤버 데이터를 private 영역에 놓고, public 함수를 통해서 접근하는 것**
- **상속성 : 기존 클래스의 특징을 물려 받아서 새로운 특징을 추가하는 것**

입니다.

그럼, 마지막 특징인 다형성은 무엇일까요 ?

이번에는 다형성(polymorphism)에 대해서 자세히 살펴 보겠습니다.

앞에서 만든 도형편집기에서 아래 코드를 생각해 봅시다.

```
for (auto p : v)
    p->Draw();  // 이 순간 p가 어떤 객체를 가리키는 지에 따라 다른 함수가
                // 호출됩니다.
```

"p->Draw()" 와 같이 동일한 함수를 호출하는 것 같은 표현이 상황(p 가 가리키는 객체가 실제로 어떤 타입인가 ?)에 따라 다른 함수를 호출하게 되는 것을 **"다형성(Polymorphism)"** 이라고 합니다.

어떤 장점이 있을까요 ? 앞장에서 만든 도형 편집기는 두개의 도형(Rect, Circle)만을 다룰수 있습니다. 그런데, 시간이 지나서 나중에 새로운 도형인 "Triangle"을 추가 했다고 생각해 봅시다.

```
class Triangle : public Shape
{
public:
    virtual void Draw() { cout << "Triangle Draw" << endl; }
};
```

이제 새롭게 추가된 Triangle 때문에 아래 한줄이 수정될 필요가 있을까요 ?

```
for (auto p : v)
    p->Draw();  // Triangle 클래스가 추가 되어도 이 한줄은 수정될 필요가 없습니다.
```

즉, 미래에 어떤 도형이 추가되어도

"Shape 클래스로부터 파생되고, Draw()가상 함수를 재정의 했다면"

기존에 있는 "p->Draw()" 코드는 수정될 필요가 없습니다. 아주 좋은 장점 입니다.

이런 특징은 "OCP" 라고 합니다.

▪ 개방 폐쇄의 법칙(Open Close Principle, OCP)

객체지향 S/W 설계의 가장 중요한 원칙 중의 하나는

"기능 확장에 열려있고(Open, 미래에 새로운 코드가 추가되어도)"

"코드 수정에 닫혀있어야(Close, 기존 코드는 수정되면 안된다)는 원칙(Princple) 입니다."

대부분의 객체지향 디자인은 "OCP" 라는 원칙을 지키기 위한 코딩 기법입니다. OCP 외에도 객체지향 설계원칙은 총 5 가지가 있습니다. 흔히, "SOLID"라고 이야기 합니다.

"SOLID(SRP, OCP, LSP, ISP, DIP)"

객체지향 설계에 대한 자세한 내용은 "객체지향 디자인 패턴" 과정을 수강하시면 됩니다.

20 장. 함수 바인딩(function binding) 과 가상함수의 원리

20.1 함수 바인딩 (Function Binding) 개념

함수를 호출하는 코드가 있을때, 실제 어떤 함수를 호출 할 것인가를 결정하는 것을 함수 바인딩(Function Binding) 이라고 합니다.

```
int main()
{
    Animal* p = new Dog;
    p->Cry();   // 이순간 Animal의 Cry()를 호출할지, Dog의 Cry()를 호출할지
                // 결정해야 합니다.
}
```

함수 바인딩에는 크게 2 가지 방법이 있습니다.

static binding	어떤 함수를 호출할지를 컴파일러가 컴파일 시간에 결정합니다. 컴파일 시간에 호출을 결정하므로 오버헤드가 없습니다. early binding 이라고도 합니다. C++ 비가상 함수
dynamic binding	어떤 함수를 호출할지를 실행시간에 결정합니다. 실행시간에 실제 객체가 무엇인가를 조사한후 함수를 호출해야 하므로 호출에 약간의 오버헤드가 있습니다. late binding 이라고도 합니다. C++ 가상함수, java 의 멤버함수, objective-c 의 멤버 함수

결국, C++에서 가상함수는 dynamic binding 을 하고, 가상함수가 아닌 경우는 static binding 을 하게 됩니다. java, objective-c 등의 C++외의 대부분 다른 언어는 dynamic binding 을 합니다. 결국, java 는 모든 함수가 가상함수라는 이야기 입니다.

지금부터 두가지 바인딩에 대해 자세히 살펴 보겠습니다.

20.2 static binding

static biding 은 어떤 함수를 호출 할 지를 컴파일러가 컴파일 시간에 결정해야 합니다. 그런데, 중요한 것은 컴파일 시간에는 실제 객체가 어떤 객체인지 모른다는 점입니다.

- **"컴파일 시간에는 실제 객체가 어떤 타입인지 알 수 없다."**

아래 코드를 잘생각해 봅시다. 주석을 주의 깊게 읽어 보세요

```
int main()
{
    int n = 0;
    Animal* p = 0;  // Animal뿐 아니라 모든 종류의 Animal의 파생 클래스 주소를
                    // 담을수 있습니다.
    cin >> n;

    // 사용자의 입력 결과에 따라 생성되는 객체의 타입이 다릅니다.
    if (n == 1)
        p = new Animal;
    else
        p = new Dog;

    // 아래 코드를 컴파일 할때 p가 가리키는 객체가 무슨 타입인지 컴파일러가
    // 알수 있을까요 ?
    p->Cry();
}
```

위 코드는 사용자의 입력에 따라 생성되는 객체의 종류 달라 집니다. 그런데, 사용자에게 입력을 받는 것은 실행시간의 개념 이므로 컴파일 시간에는 실제 어떤 타입의 객체가 생성 될지는 알수는 없습니다.

- **"컴파일가 아닌 것은 객체의 타입이 아닌 포인터의 타입이다."**

결국 컴파일러가 아는 것은 p 의 타입이 "Animal*" 라는 정보 밖에 없습니다. 그래서, 컴파일 시간에 함수 호출을 결정하면 무조건 포인터 타입만 보고 함수 호출을 결정합니다. 결국 Animal 이 Cry 가 호출됩니다. C++에서는 가상함수가 아닌 경우 static binding 을 하게 됩니다. 따라서, 실제 객체의 종류에 상관 없이 포인터 타입만 보고 호출합니다.

```
Animal* p =  new Dog;
p->Cry();   // Cry 함수가 가상함수가 아닌 경우 실제 객체(Dog)가 아닌
            // 포인터 타입(Animal*)의 Cry 함수가 호출됩니다.
```

20.3 dynamic binding

C++에서 멤버 함수를 가상함수로 만들면 함수 호출을 결정할 때 dynamic binding 을 하게 됩니다. Cry 함수가 가상 함수면 다음 처럼 컴파일/실행 하게 됩니다.

```
Animal* p =  new Dog;
p->Cry();   // 1. 컴파일 할 때 p가 가르키는 메모리를 조사하는 기계어 코드를 생성합니다.
            // 2. 실행시간에 메모리를 조사한후 함수 호출을 결정합니다.
```

따라서, 가상함수는 일반 멤버 함수보다 호출시에 약간의 성능저하가 따르게 됩니다. 지금 부터, 가상함수의 원리에 대해서 좀더 자세히 살펴 보겠습니다.

- **가상함수 테이블**

간단한 코드를 가지고 가상함수의 원리에 대해서 자세히 살펴 보겠습니다.

```
class Animal
{
    int age;
public:
    // 2개의 가상함수가 있습니다.
    virtual void Cry() {}
    virtual void Run() {}
};
class Dog : public Animal
{
    int color;
public:
    // 파생클래스에서 2개의 가상함수중 한개만 override 했습니다.
    virtual void Cry() {}
};
```

[참고] 기반 클래스인 Animal의 소멸자는 반드시 가상 소멸자로 만들어야 합니다. 코드를 단순화 해서 설명하기 위해 생략 했습니다.

위 코드를 컴파일 하면 컴파일러는 다음과 같은 의미를 가지는 코드를 생성하게 됩니다. 아래 코드에서 볼수 있는 RTTI 정보는 다음 장에서 배우게 됩니다.

```
// 1. Animal 의 모든 가상함수의 주소를 담은 배열을 생성합니다
void* Animal_vtable[] = {  Animal 의 RTTI 정보, &Animal::Cry, &Animal::Run };

class Animal
{
    void* vtptr;    // 2. 가상 함수 테이블을 가리키기 위한 포인터를 추가합니다.
    int age;
public:
    Animal() : vtptr(Animal_vtable) {} // 3. 생성자에서 vtptr을 초기화 합니다.
    virtual void Cry() {}
    virtual void Run() {}
};
// Dog 의 vtable의 경우 재정의하지 않은 가상함수는 기반 클래스 함수 주소가 놓이게
됩니다.
void* Dog_vtable[] = {  Dog 의 RTTI 정보, &Dog::Cry, &Animal::Run };

class Dog : public Animal
{
    void* vtptr;
    int color;
public:
    Dog() : vtptr(Dog_vtable) {}
    virtual void Cry() {}
};
```

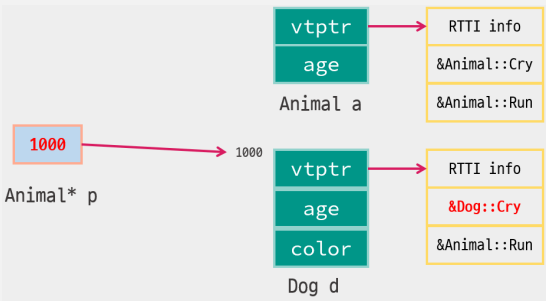
[참고] 가상함수 테이블을 나타내는 배열의 타입을 void*로 했지만, 단지 함수의 주소를 담는 것을 표현한 것 뿐입니다. 모든 C++컴파일러가 이런 방식으로 가상함수를 처리하는 것은 아닙니다. 대표적인 방식을 설명하는 의사코드 입니다.

결국 가상함수를 사용하는 클래스의 객체를 생성하면 다음과 같이 메모리에 놓이게 됩니다.

사용자가 작성한 코드

```
int main()
{
    Animal a;
    Dog    d;

    Animal* p = &d;
    p->Cry();
}
```

메모리



그리고, "p->Cry()" 코드를 컴파일 할 때 컴파일러는 다음 처럼 동작 합니다.

① **컴파일 시간에는 p 가 Animal* 라는 정보 밖에 알지 못하므로 Animal 클래스의 조사 해서 Cry 함수가 가상함수 인지 아닌지를 판단합니다.**

② **가상함수가 아니라면 무조건 Animal 의 Cry 를 호출합니다.**

③ **가상함수라면 p 가 가르키는 곳을 참조 해서 가상함수 테이블의 주소를 꺼내고, Cry 는 순서상 첫번째 가상함수 이므로 배열의 첫번째 값을 꺼내서 호출하는 기계어 코드를 생성해 놓습니다**

④ **프로그램이 실행할 때 3 번에서 실행된 기계어 코드를 실행하므로 Dog 의 Cry 가 호출됩니다.**

▪ 가상 함수를 가지는 객체의 크기

어떤 클래스가 가상함수를 가지면 컴파일러에 의해 추가된 가상함수 테이블을 가르키는 포인터 때문에 객체의 크기는 4 바이트(64 비트 환경에서는 8 바이트)가 더 커지게 됩니다.

사용자가 만든 코드

```
class Animal
{
    int age;
public:
    virtual void Cry() {}
    virtual void Run() {}
};

int main()
{
    Animal a;
    cout << sizeof(a) << endl; // 8
}
```

컴파일러에 의해 생성된 코드

```
class Animal
{
    void* vtptr;    // 컴파일러에 의해 추가
                    // 됩니다.
    int age;
public:
    // ……
};
```

▪ 가상함수의 단점

결국 가상함수를 사용하면 "실행시간에 다형성(polymorphism)" 을 사용할수 있지만, 다음과 같은 단점도 있습니다.

① **가상함수 테이블 때문에 메모리 오버헤드가 있습니다. (가상함수가 많거나 상속의 계층이 깊어지만 주의 해야 합니다.)**

② 객체를 만들 때 vtptr 멤버가 추가 되므로 객체 크기가 4 바이트(8 바이트)증가 합니다.

③ 실제 함수를 호출할 때 가상함수 테이블에서 주소를 꺼내서 호출하므로 일반 호출보다 약간 느립니다.

④ 가상함수는 실행시간에 어느 함수를 호출하지를 결정하므로 인라인 함수가 될수 없습니다.

21 장. 추상 클래스와 인터페이스

21.1 순수 가상함수와 추상 클래스

가상함수를 만들 때 함수의 구현부가 없고 선언부 뒤에 "=0"으로 표기하는 것을 "순수 가상함수(pure virtual function)" 라고 합니다. 또한, 순수 가상함수를 한 개 이상 가지고 있는 클래스를 추상 클래스(Abstract Class) 라고 합니다.

```
// 순수 가상함수가 한개 이상 있는 클래스를 추상 클래스라고 합니다.
class Shape
{
public:
    virtual void Draw() = 0;    // 순수 가상 함수 ( pure virtual function )
                                // 특징 1. 구현부가 없습니다.
                                // 특징 2. 함수 선언뒤에 "=0" 을 표시 합니다.
};
```

이와 같은 추상클래스는 어떤 특징이 있고 왜 사용 할까요 ? 지금부터 자세히 살펴 보도록 하겠습니다.

- **추상 클래스는 객체를 만들 수 없습니다.**

추상 클래스는 순수 가상함수를 가지고 있습니다. 그런데, 순수 가상함수는 구현이 없으므로 함수를 사용할 수가 없습니다. 그래서 추상 클래스는 객체 자체를 만들 수 없습니다. 하지만 포인터는 만들 수 있습니다.

```
int main()
{
    Shape  s; // error. 추상 클래스는 객체를 만들 수 없습니다.
    Shape* p; // ok.    포인터는 만들 수 있습니다.
}
```

- **추상 클래스와 파생 클래스**

아래 코드에서 Rect 는 추상 클래스 일까요 ? 아닐까요 ?

```
class Shape
{
public:
    virtual void Draw() = 0;
};
class Rect : public Shape
{
};
int main()
{
    Rect r; // ?
}
```

Rect 클래스는 멤버가 없는 것처럼 보이지만 기반 클래스로부터 순수 가상함수인 Draw()를 상속 받게 됩니다. 따라서 Rect 로 역시 추상 클래스 입니다. 그래서 Rect 역시 객체를 만들 수 없습니다.

```
Rect r; // Error. Rect 도 추상 클래스 입니다.
```

그렇다면, Rect 를 사용하려면 어떻게 해야 할까요 ? 기반 클래스가 가진 순수 가상함수인 Draw()함수의 구현부를 제공해야만 Rect 를 사용할 수 있습니다.

```
class Rect : public Shape
{
public:
    virtual void Draw() { cout << "Rect Draw" << endl; }
};
int main()
{
    Rect r; // ok. 기반 클래스로 부터 상속 받은 순수 가상함수인
            // Draw의 구현부를 제공했으므로 Rect는 더이상 추상클래스가 아닙니다.
}
```

결국, 추상 클래스를 만드는 의도는

"파생 클래스에게 특정 멤버함수를 반드시 만들라고 지시"

할 때 사용하기 위해서 입니다.

참고로 Draw 함수를 순수 가상함수가 아닌 일반 가상함수를 사용하면 파생 클래스에서는 재정의를 해도 되고 안해도 됩니다. 하지만, 순수 가상함수로 제공하면 반드시 재정의 해야 합니다.

```
class Shape
{
public:
    // 순수 가상함수가 아닌 일반 가상함수일 경우
    virtual void Draw() {}
};
class Rect : public Shape
{
};
int main()
{
    Rect r; // ok. Rect에서 Draw의 구현부를 제공하지 않아도 Rect를 사용할 수 있습니다.
}
```

결국, 다양한 도형 타입을 만들 때,

"모든 도형은 반드시 Draw 함수가 있어야 된다."

는 규칙을 만들려면 Shape 클래스안에 Draw 함수를 순수 가상함수로 제공하면 됩니다.

이런 추상 클래스는 결국 "인터페이스"라는 개념으로 발전됩니다. 다음장에서는 인터페이스의 개념에 대해서 자세히 살펴 보겠습니다.

21.2 인터페이스와 결합도(coupling)

요즘 자동차에는 블랙박스가 달려 있어서 차량주변을 상시 녹화 할 수 있습니다. 블랙박스용 카메라와 자동차를 생각해 봅시다.

```
class Camera
{
public:
    void startRecord() { cout << "starting Record" << endl; }
    void stopRecord()  { cout << "stop Record" << endl; }
};
class Car
{
    Camera* pCamera = 0;
public:
    void setBlackBoxCamera(Camera* p) { pCamera = p; }
    void startBlackBoxCamera()        { pCamera->startRecord(); }
};
int main()
{
    Camera cam;
    Car car;
    car.setBlackBoxCamera(&cam);
    car.startBlackBoxCamera();
}
```

위 코드는 아무 문제없이 잘 실행 됩니다. 그런데, 세월이 지나서 좀더 좋은 카메라를 만들었다고 생각해 봅시다.

```
class HDCamera
{
public:
    void startRecord() { cout << "starting Record in HD" << endl; }
    void stopRecord()  { cout << "stop Record in HD" << endl; }
};
```

이제, 새롭게 만든 "HDCamera" 클래스를 Car 에 연결할 수 있을까요 ? 안됩니다. 그 이유는 Car 클래스는 내부적으로 "Camera" 타입만 사용하므로 "HDCamera"는 사용할 수 없습니다.

이처럼 하나의 클래스가 다른 클래스를 사용할 때 클래스의 이름을 직접 사용하는 것을 보통 "강한 결합(tightly coupling)" 이라고 합니다.

```
class Car
{
    Camera* pCamera = 0;
public:
    void setBlackBoxCamera(Camera* p)   // 강한 결합 ( tightly coupling )
    {                                   // Car 는 항상 Camera 만 사용해야 합니다.
        pCamera = p;                    // 다른 카메라(HDCamera)로 교체할 수가
                                        // 없습니다.
    }
};
```

강한 결합은 교체가 불가능 하고 확장성이 없습니다. 위 코드에서 Car 클래스는 Camera 만 사용할 수 있고 다른 카메라는 사용할 수 없게 됩니다.

확장성 있는 시스템을 만들려면 어떻게 해야 할까요 ?

- **인터페이스와 약한 결합 (loosely coupling)**

Car 클래스나 Camera 클래스를 만들기 전에, 자동차에 연결하기 위해 카메라가 지켜야 하는 규칙을 먼저 설계해 봅시다. 먼저 IBlackBoxCamera 라는 추상 클래스를 만들어 봅시다.

```
class IBlackBoxCamera
{
public:
    virtual void startRecord() = 0;
    virtual void stopRecord()  = 0;
    virtual ~IBlackBoxCamera() {}
};
```

그리고,

"자동차에 연결하기 위한 모든 카메라는 IBlackBoxCamera 추상 클래스로부터 파생 되어야 한다."

라는 규칙을 만들어 봅시다.

결국, "모든 카메라는 IBlackBoxCamera 로부터 파생되어야 한다"라는 의미는 모든 카메라는 반드시 startRecord 함수와 stopRecord 함수가 있어야 한다는 의미 입니다.

아직, 실제 카메라는 없고 규칙만 있습니다. 하지만, 자동차를 만들 때는 실제 카메라가 없어도 규칙이 있으므로 규칙대로만 사용하면 됩니다. 추상 클래스는 객체를 만들 수는 없지만 포인터는 만들수 있으므로 Car 클래스는 아래 처럼 만들게 됩니다.

```
class Car
{
    IBlackBoxCamera* pCamera = 0;
public:
    void setBlackBoxCamera(IBlackBoxCamera* p)  { pCamera = p; }
    void startBlackBoxCamera()                  { pCamera->startRecord(); }
};
```

여기 까지 에서 Car 가 사용하는 카메라는 특정 클래스로 확정된 것이 아니라, IBlackBoxCamera 라는 추상 클래스로부터 파생된 모든 클래스는 전부 사용할 수 있습니다.

이제, 실제 카메라를 만들어 봅시다. 단, 모든 카메라는 반드시 규칙을 지켜야 합니다.

```
class Camera : public IBlackBoxCamera
{
public:
    void startRecord() { cout << "starting Record" << endl; }
    void stopRecord()  { cout << "stop Record" << endl; }
};
```

이제, Camera 클래스는 Car 에 연결될 수 있습니다. 뿐만 아니라 어떤 종류의 카메라 클래스라도 규칙인 "IBlackBoxCamera 로 부터 파생되어야 한다" 만 지키면 Car 클래스와 연결 할 수 있습니다.

이처럼 Car 가 Camera 를 사용하지만 Camera 라는 이름을 직접 사용하지 않고 기반 클래스인 IBlackBlockBox* 를 사용해서 Camera 의 기능을 사용하게 되는 것은 **"약한결합(loosely coupling)"** 이라고 합니다. 약한 결합을 사용하면 교체 가능한 확장성 있는 디자인을 할 수 있습니다.

다음은 완성된 코드 입니다.

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

class IBlackBoxCamera
{
public:
    virtual void startRecord() = 0;
    virtual void stopRecord()  = 0;
    virtual ~IBlackBoxCamera() {}
};
class Camera : public IBlackBoxCamera
{
public:
    void startRecord() { cout << "starting Record" << endl; }
    void stopRecord()  { cout << "stop Record" << endl; }
};
class HDCamera : public IBlackBoxCamera
{
public:
    void startRecord() { cout << "starting Record in HD" << endl; }
    void stopRecord() { cout << "stop Record in HD" << endl; }
};
class Car
{
    IBlackBoxCamera* pCamera = 0;
public:
    void setBlackBoxCamera(IBlackBoxCamera* p) { pCamera = p; }
    void startBlackBoxCamera() { pCamera->startRecord(); }
};
int main()
{
    // Car는 다양한 종류의 카메라를 사용할 수 있습니다
    // 단, 카메라는 지켜야 하는 규칙이 있습니다.
    //Camera cam;
    HDCamera cam;

    Car car;
    car.setBlackBoxCamera(&cam);
    car.startBlackBoxCamera();
}
```

- **용어 정리**

위 코드에서 카메라가 지켜야 하는 규칙을 담은 추상 클래스인 "IBlackBoxCamera"를 보통 **"인터페이스(interface)"** 라는 용어를 사용합니다. java, C#등에는 별도로 interface 키워드가 있지만 C++에서는 "interface'라는 키워드나 문법은 없고 추상 클래스를 사용해서 인터페이스를 만들게 됩니다.

그리고, 인터페이스를 만들 때는 일반적으로 순수 가상 함수를 public 에 놓는 경우가 대부분 이므로 struct 를 사용하는 경우도 많이 있습니다.

```
struct IBlackBoxCamera  // 구조체는 접근 지정자 생략시 기본적으로 public 입니다.
{
    virtual void startRecord() = 0;
    virtual void stopRecord()  = 0;
    virtual ~IBlackBoxCamera() {}
};
```

또한, 모든 카메라를 만들 때 는 startRecord/stopRecord 함수를 반드시 만들어야 합니다. 그래서, 규칙을 표현 할 때는

"모든 카메라는 IBlackBoxCamera 로부터 파생되어야 한다"

라고 하는 것 보다는

"모든 카메라는 IBlackBoxCamera 인터페이스를 구현해야 한다."

라고 표현하는 것이 좋습니다.

결국, 객체지향 디자인은 인터페이스 기반의 약한 결합으로 확장성 있고 교체 가능한 유연한 설계를 목표로 하고 있습니다. 다양한 객체지향 디자인의 기술은 이 책의 범위를 넘어 가므로 "객체지향 디자인과 라이브러리 설계의 기술" 과정을 참고 하시기 바랍니다.

22 장. RTTI(Run-Time Type Information)

22.1 RTTI 개념.

Animal* 타입의 변수 p 가 있을 때, p 에는 Animal 의 주소 뿐 아니라 Animal 의 모든 파생 클래스의 주소도 담을수 있습니다. 이때, p 가 실제로 어떤 객체를 가르키는지 조사하는 것을 RTTI (Run-Time Type Information) 라고 합니다.

```
void foo(Animal* p)
{
    // RTTI :  p가 실제로 어떤 타입의 객체를 가리키는지 조사하는것
}
```

RTTI 는 가상함수 테이블로 관리되는 type_info 객체가 핵심입니다.

- **typeid()와 type_info 객체**

클래스가 가상함수를 가지면 객체를 생성시 가상함수 테이블이 만들어 집니다. 가상함수 테이블 안에는 가상함수의 주소 뿐 아니라 클래스의 정보를 담은 type_info 객체가 있습니다.

특정 포인터 p 가 실제 어떤 객체를 가르키는지 알고 있으면 type_info 객체를 꺼내면 됩니다.

type_info 객체를 꺼내려면

① **typeid(객체) 또는 typeid(클래스이름)으로 꺼내면 됩니다.**

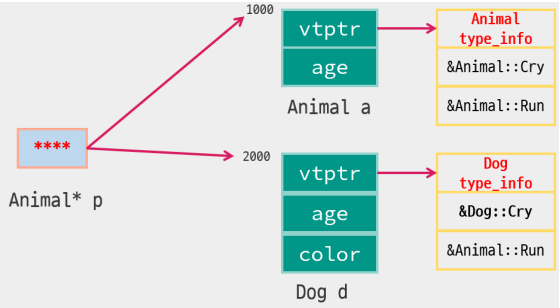
② **type_info 객체를 꺼낼때면 반드시 const reference 로 받아야 합니다**

③ **RTTI 기술을 사용하려면 <type_info> 헤더를 포함해야 합니다.**

다음 코드는 p 가 가르키는 객체의 type_info 를 꺼내는 코드 입니다.

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

void foo(Animal* p)
{
    const type_info& t = typeid(*p);
```

```
    cout << t.name() << endl;
}

int main()
{
    Animal a;
    foo(&a);
    Dog d;
    foo(&d);
}
```



[참고] Animal 과 Dog 클래스는 앞장에서 만든 코드와 동일 합니다

결국 위 코드는 p 가 실제 어떤 객체인지에 따라 Animal 또는 Dog 의 type_info 를 꺼내게 됩니다.

위 코드를 실행하면 결과는 다음과 같습니다.

```
class Animal;
class Dog;
```

[참고] g++로 실행하면 클래스 이름이 다른 형태로 나오게 됩니다.

그런데, 대부분 필요한 것은 클래스 이름을 출력하는 것이 아니라 p 가 가르키는 객체가 어떤 타입인지를 조사하는 것입니다. 아래 처럼 만들면 됩니다.

```
void foo(Animal* p)
{
    const type_info& t1 = typeid(*p);  // 객체 이름으로 type_info를 꺼냅니다.
    const type_info& t2 = typeid(Dog); // 클래스 이름으로 type_info를 꺼냅니다.

    // p가 Dog를 가르킨다면 t1과 t2는 같은 type_info를 가르키게 됩니다.
    if (t1 == t2)
    {
        // p를 Dog 타입으로 캐스팅해서 사용하면 됩니다.
        Dog* pDog = static_cast<Dog*>(p);
        //......
    }
}
```

실전에서는 아래 처럼 간단하게 사용하면 됩니다.

```
if (typeid(*p) == typeid(Dog))
{
    Dog* pDog = static_cast<Dog*>(p);
    //......
}
```

- **가상함수가 없는 경우**

가상함수가 없는 타입의 경우에는 포인터 타입으로 type_info 객체를 만들게 됩니다.

```
int main()
{
    Animal* p = new Dog;

    // Animal에 가상함수가 없으면 Animal의 type_info
    // 가상함수가 있으면 Dog 의 type_info가 나오게 됩니다.
    const type_info& t = typeid(*p);

    cout << t.name() << endl;
}
```

22.2 dynamic_cast

- static_cast 의 단점

기반 클래스 포인터에는 기반 클래스 뿐 아니라 파생 클래스도 담을 수 있습니다. 그렇다면 기반 클래스의 포인터를 다시 파생 클래스로 포인터 타입으로 명시적 캐스팅을 하면 어떨 까요 ?

```
void foo(Animal* p)
{
    // 컴파일 시간에 캐스팅 하므로 객체를 조사할수 없습니다.
    // p가 Animal를 가리킬때도 주소가 나오게 됩니다.
    Dog* pDog = static_cast<Dog*>(p);

    cout << pDog << endl; // 항상 주소가 나오게 됩니다.
}
int main()
{
    Animal a;
    foo(&a);

    Dog    d;
    foo(&d);
}
```

이때, p 가 실제로 Dog 를 가리키고 있다면 문제가 없지만 p 가 Animal 을 가리키고 있다면 문제가 될수 있습니다. 하지만, static_cast 는 "static" 이라는 이름에서 알수 있듯이 컴파일 시간에 캐스팅을 수행하므로 p 가 실제로 Animal 을 가리키는지 Dog 를 가리키는지 알 수 없습니다.

그래서 무조건 캐스팅에 성공하게 됩니다. 즉, pDog 에는 항상 유효한 주소가 들어오게 됩니다. p 가 실제로 Dog 를 가리킬 때만 캐스팅에 성공하고 p 가 Dog 가 아닌 객체(Animal 또는 다른 동물)을 가리킬때는 캐스팅에 실패 하도록 할수 없을 까요 ?

- dynamic_cast

p 가 실제로 Dog 를 가리킬 때는 캐스팅에 성공되게 하려면 컴파일 시간이 아닌 실행 시간에 캐스팅을 수행해야 합니다.

dynamic_cast 를 사용하면 실행시간에 캐스팅을 수행할 수 있기 때문에 p 가 가르키는 실제 가리키는 객체를 조사할수 있습니다. p 가 가리키는 곳이 Dog 가 아니면 0 을 리턴하게 됩니다.

```
void foo(Animal* p)
{
    // p가 Dog를 가리키는 경우만 성공하고 그렇지 않으면 0을 리턴합니다.
    Dog* pDog = dynamic_cast<Dog*>(p);

    cout << pDog << endl;
}
```

주의 할점은 dynamic_cast 를 사용하려면 반드시 가상함수가 한 개이상 있어야 합니다. 가상함수가 없을 경우 dynamic_cast 는 runtime error 가 발생하게 됩니다.

또한, dynamic_cast 는 실행시간에 수행되므로 컴파일 시간에 캐스팅이 수행되는 static_cast 보다는 느립니다. p 가 실제 어떤 객체를 가리키는지를 정확히 알고 있을 때는 dynamic_cast 를 사용하지 말고 static_cast 를 사용하는 것이 좋습니다.

- RTTI 요점 정리

결국 실행시간에 타입을 조사하려면 아래의 2 가지 방법을 사용할수 있습니다.

① typeid()

② dynamic_cast<>

typeid()를 사용하는 방법

```
void foo(Animal* p)
{
    if (typeid(*p) == typeid(Dog))
    {
        // p가 Dog를 가리키는 것이
        // 확실하므로 static_cast를
        // 사용하는 것이 좋습니다.
        Dog* pDog = static_cast<Dog*>(p);
    }
}
```

dynamic_cast<> 를 사용하는 방법

```
void foo(Animal* p)
{
    Dog* pDog = dynamic_cast<Dog*>(p);

    if ( pDog != 0 )
    {

    }
}
```

SECTION 4

Operator Overloading

C++에서는 사용자가 만든 타입에 대해서도 +, - 등의 연산을 적용 할 수 있습니다.

"사용자 정의 타입도 표준 타입 처럼 동작해야 한다." 는 철학에서 출발한 연산자 재정의는 +, - 같은 연산자도 함수로 표현 할 수 있는 문법입니다.

이번 장에서는 연산자 재정의 관련 문법을 배우게 됩니다. 또한, 연산자 재정의 문법을 활용한, 함수 객체, 스마트 포인터 등의 개념도 배우게 됩니다.

이번 장에서 다음과 같은 개념을 배우게 됩니다.

"연산자 재정의 개념, cout과 ostream, 증가 연산자 재정의, 함수 객체, 스마트 포인터, 대입연산자, String 클래스 만들기"

23 장. 연산자 재정의 개념

23.1 연산자 재정의 개념

C/C++ 에서 2 개의 정수 및 실수를 더 할 때는 "+" 연산자를 사용합니다. 그렇다면 사용자 정의 타입인 Point 객체 2 개를 더할 수 있을까요 ?

```
Point p1(1, 1);
Point p2(2, 2);

int n = 1 + 2;      // 정수를 더할 때는 '+' 연산자를 사용합니다.
Point p3 = p1 + p2; // error. 어떻게 더할지는 알 수 없습니다.
```

당연이 컴파일러는 사용자가 만든 타입인 Point 를 어떻게 더해야 할지 알 수 없기 때문에 에러가 발생합니다. 하지만, C++에서는 +, - 같은 연산자도 함수로 만들 수 있습니다. 컴파일러는 "p1 + p2" 구문을 만나면 다음 처 럼 해석 하게 됩니다.

```
Point p3 = p1 + p2;    // "operator+(p1, p2)" 함수를 찾게 됩니다.
```

따라서, Point 타입의 객체 2 개를 더할 수 있게 하려면 Point 타입 2 개를 인자로 가지는 "operator+" 함수를 제공하면 됩니다. 또한, 멤버 함수가 아닌 일반 함수에서 Point 의 private 멤버인 x, y 를 접근하고 있으므로 friend 함수로 등록해야 합니다. 다음은 완성된 코드 입니다.

```
class Point
{
    int x, y;
public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    friend Point operator+(const Point& p1, const Point& p2);
};
Point operator+(const Point& p1, const Point& p2)
{
    Point ret(p1.x + p2.x, p1.y + p2.y);
    return ret;
}
int main()
```

```
{
    Point p1(1, 1);
    Point p2(2, 2);
    Point p3 = p1 + p2;
}
```

23.2 일반 함수와 멤버 함수

연산자 재정의 함수는 일반 함수 뿐 아니라 멤버 함수로도 구현할 수 있습니다.

컴파일러는 "p1 + p2" 의 코드를 만날 때 다음 처럼 해석하게 됩니다.

① p1.operator+(p2) 으로 해석합니다. 즉, p1의 멤버 함수중 operator+()함수를 찾게 됩니다.

② operator+(p1, p2) 로 해석합니다. 멤버함수가 없을 경우 멤버함수가 아닌 일반 함수 operator+() 함수를 찾게 됩니다.

이항 연산자의 경우 멤버가 아닌 일반 함수(friend 함수)로 구현하면 함수 인자가 2 개 이지만 멤버 함수로 구현하면 함수의 인자는 한 개가 됩니다.

✍ 멤버 함수로 구현한 operator+()

```
class Point
{
    int x, y;
public:
    Point(int a = 0, int b = 0)
            : x(a), y(b) {}

    // 이항연산자를 멤버함수로 구현할
    // 경우 함수인자는 한 개가 됩니다.
    Point operator+(const Point& p);
};
Point Point::operator+(const Point& p)
{
    Point ret(x + p.x, y + p.y);
    return ret;
}
```

✍ 일반 함수(friend)로 구현한 operator+()

```
class Point
{
    int x, y;
public:
    Point(int a = 0, int b = 0)
        : x(a), y(b) {}
    friend Point operator+(const Point& p1,
                           const Point& p2);
};
// 이항연산자를 일반 함수로 구현하면
// 함수인자는 2개가 됩니다.
Point operator+(const Point& p1,
                const Point& p2)
{
    Point ret(p1.x + p2.x, p1.y + p2.y);
    return ret;
}
```

하나의 클래스에 대해 operator+() 함수를 멤버함수와 일반함수 모두를 제공하면 멤버함수를 우선적으로 사용하게 됩니다. 일반적으로는 둘 중 하나만 제공해야 합니다.

그렇다면, operator+()함수를 만들 때 멤버 함수로 만드는 것이 좋을까요 ? 아니면 일반함수로 만드는 것이 좋을까요 ? 다양한 관점에서 여러 주장이 있을 수 있지만, 일반적으로 객체의 상태가 변경되는 경우 멤버함수로, 객체의 상태가 변하지 않은 경우 일반 함수로 만드는 경우가 많이

있습니다. 예를 들면 "++a"는 실행 후 a 의 값이 변경되므로 ++ 연산자는 멤버 함수로, "a + b"는 실행 후 a 값이 변하지 않으므로 일반 함수로 만드는 경우가 많이 있습니다. 하지만 절대적인 규칙은 아니고, 다른 주장도 많이 있습니다.

23.3 연산자 재정의 주의 사항

연산자 재정의시 몇가지 주의할 점이 있습니다.

- **연산자 재정의시 한 개 이상의 인자는 반드시 사용자 정의 타입이 되어야 합니다.**

인자로 int 타입 2 개를 가지는 operator+을 만들 수는 없습니다. 반드시, 함수 인자 중 한 개는 사용자 정의 타입이 되어야 합니다.

```
int operator+(int a, int b);        // error. 2개인자가 모두 primitive 타입입니다.
int operator+(int a, double b);     // error. 2개인자가 모두 primitive 타입입니다.
Point operator+(int a,   Point b);  // ok.
Point operator+(Point a, Point b);  // ok.
```

- **인자의 개수를 변경할 수는 없습니다.**

이항 연산자를 일반 함수로 만들 때는 인자가 2 개, 멤버 함수로 만들 때는 인자가 1 개 있어야 합니다. operator+() 함수를 만들 때 인자를 3 개로 만들 수는 없습니다.

```
Point operator+(Point a, Point b, Point c);  // error
```

- **연산자의 우선순위를 변경할 수 없습니다.**

곱하기(*) 연산은 더하기(+) 연산 보다 우선 순위가 높습니다. 아무리 연산자 재정의 해도 우선순위를 변경할 수는 없습니다.

- **새로운 연산자를 만들 수는 없습니다.**

```
Point operator+*(Point a, Point b);  // error.  +* 라는 연산자는 없습니다.
```

- :: . .* ?: 연산자는 오버로딩 할 수 없습니다.

```
Point operator::(Point a);  // error.  :: 연산자는 재정의 불가능 합니다.
```

- =, (), [], -> 연산자는 멤버 함수로만 만들 수 있습니다.

```
class Vector
{
public:
    int& operator[](int idx);       // ok, [] 연산자는 멤버 함수로 만들 수 있습니다.
};
int& operator[](Vector& v, int idx); // error. [] 연산자는 일반함수로 만들 수 없습니다.
```

C++ 은 거의 대부분의 연산자를 재정의 할 수 있습니다. 하지만, 어떤 연산자는 재정의 했을 때 득보다는 실이 많은 경우도 있습니다. 반대로, 어떤 연산자는 재정의시에 아주 유용하게 사용되는 경우도 있습니다. 다양한 연산자 재정의를 활용한 기술 중에 중요한 것 위주로 살펴 보도록 하겠습니다.

24 장. cout 과 ostream

24.1 cout 의 원리 - “<<” 연산자 재정의

printf 함수와 다르게 cout 은 출력하려는 값이 정수 인지 실수인지를 표시할 필요가 없습니다. 그렇다면 과연, cout 의 정체는 무엇일까요 ?

cout 은 결국 객체 입니다. 아래 코드를 살펴 봅시다.

```
int main()
{
    int    n = 10;
    double d = 3.4;

    cout << n;  // 원리는 cout.operator<<( n) 입니다.
    cout << d;  //

    // 아래처럼 직접 사용해도 됩니다.
    cout.operator<<(n); // cout.operator( int )    함수를 호출합니다.
    cout.operator<<(d); // cout.operator( double ) 함수를 호출합니다.
}
```

cout 은 ostream 이라는 클래스 타입의 객체입니다. 이해를 돕기 위해 ostream 클래스와 유사한 코드를 직접 만들어 보도록 하겠습니다.

```
#include <stdio.h>

namespace std
{
    class ostream
    {
    public:
        // 모든 기본 타입에 대해서 "<<" 연산자를 재정의 합니다.
        // 리턴 값을 주의 깊게 보세요
        ostream& operator<<(int    n) { printf("%d", n); return *this; }
        ostream& operator<<(double d) { printf("%f", d); return *this; }
        ostream& operator<<(char   c) { printf("%c", c); return *this; }
    };
    ostream cout;
}
```

```
int main()
{
    std::cout << 3;    // cout.operator<<( int ) 함수를 호출합니다.
    std::cout << '\n'; // cout.operator<<( char ) 함수를 호출합니다.

    // << 연산자가 자기 자신을 리턴하므로 아래 처럼 계속 이어서
    // 사용할수도 있습니다.
    std::cout << 3.4 << '\n';
}
```

[참고] cout 의 타입은 정확히 말하면 basic_ostream<> 라는 템플릿 클래스 입니다. 유니코드와 ansi 문자열을 모두 처리할 수 있도록 템플릿으로 되어 있습니다. ansi 문자열의 경우 아래 처럼 typedef 가 되어 있습니다. basic_ostream 의 내용은 상당히 복잡하므로 "Advanced C++" 과정을 참고 하시기 바랍니다.

```
typedef basic_ostream<char> ostream;
```

위 코드에서는 "<<" 연산자를 3 개의 타입에 대해서만 제공했지만 실제 표준에는 모든 기본 타입에 대해서 제공하고 있습니다. 결국 사용자가 3 을 전달하면 컴파일러는 3 이 int 라는 것을 알고 있으므로 "<<" 연산자 함수의 int 버전을 호출하기 때문에 "%d" 를 사용할 필요가 없습니다.

▪ *this 리턴의 중요성

또한, 중요한 것은 "operator<<()"함수의 리턴 값입니다. 자신 자신을 참조로 리턴 하므로 결국 리턴값은 cout 자신 입니다. 그래서, cout 을 사용할 때 << 를 계속 늘여서 사용하는 표현이 가능합니다. 다시 한번 강조하는 것은 리턴 타입은 값 타입이 아닌 참조 타입이 되어야 합니다.

24.2 사용자 정의 타입과 cout

cout 의 장점중의 하나는 정수건 실수 이건 사용자가 %d, %f 로 지정하지 않아도 cout 이 알아서 출력해준다는 점입니다. 그렇다면 다음 코드는 어떨까요 ?

```
Point pt(1, 2);
cout << pt;     // cout.operator<<(Point) 함수가 필요합니다.
```

결국 위 코드가 실행되려면 cout.operator<<(Point) 함수가 필요 합니다. 그런데, Point 클래스는 사용자가 정의 타입이므로 ostream 클래스의 멤버로 있을 수는 없습니다. 그리고 ostream 클래스는 우리가 만든 클래스가 아니므로 멤버 함수를 추가 할수도 없습니다.

해결책은 operator<< 연산자를 멤버가 아닌 일반 함수로 제공하는 것입니다.

```
cout << pt; // 1. 멤버 함수 "cout.operator<<(Point)" 가 있으면 멤버함수를 호출합니다.
            // 2. 일반 함수 "operator<<(ostream, Point)"를 호출합니다.
```

결국 사용자 타입을 cout 과 같이 사용하고 싶으면 operator<< 연산자를 멤버가 아닌 일반 함수로 제공하면 됩니다. 다음은 완성된 코드 입니다. 꼭, 실행해서 확인해 보세요

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

class Point
{
    int x, y;
public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    friend ostream& operator<< (ostream& os, const Point& pt);
};

ostream& operator<< (ostream& os, const Point& pt)
{
    return os << pt.x << ", " << pt.y;
}
int main()
{
```

```
    Point pt(1, 2);
    cout << pt << endl;
}
```

"cout << pt" 이 실행된 후에도 pt 자체는 변경되는 것이 아니므로 const 를 붙여서 전달 받습니다. 하지만 cout 은 전달 받을 때 const 를 붙이지 말아야 합니다.

- **실습 문제**

cin 은 istream 클래스의 객체 입니다. Point 타입의 변수를 cin 과 같이 사용할 수 있도록 만들어 보세요

```
int main()
{
    int n = 0;

    cin >> n;   // 이 순간  "cin.operator>>(n)" 이렇게 동작합니다.

    Point pt(0, 0);
    cin >> pt;  // 이 코드가 실행되도록 만들어 보세요
}
```

24.3 cout 과 endl

cout 은 결국 ostream 클래스의 객체입니다. 그렇다면 cout 으로 개행 을 할 때 사용하는 endl 의 정체는 무엇일까요 ? endl 은 놀랍게도 함수 입니다. 다음 두줄의 코드는 동일한 역할을 하는 코드 입니다.

```
int main()
{
    cout << endl;
    endl(cout);
}
```

그럼 지금부터 endl 의 정체에 대해 살펴 보도록 하겠습니다.

- **endl 함수 만들기**

endl 의 원리를 정확히 이해 하기 위해, endl 과 유사한 xendl 을 만들어 보도록 하겠습니다. 다음 코드를 잘 생각 해보세요. 아래 코드 에서의 cout 은 우리가 좀 전에 만든 것이 아닌 C++ 표준의 cout 입니다.

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

ostream& xendl(ostream& os)
{
    os << '\n';     // os는 결국 cout 이므로  cout << '\n' 의 코드가 됩니다.
    return os;
}

int main()
{
    xendl( cout );

    cout << xendl; // xendl은 C++ 표준 cout 과의 연동도 문제 없습니다.
}
```

xendl 은 인자로 cout 를 참조로 전달 받고 있으므로 xendl 안에서 os 는 cout 입니다. cout 은 '₩n' 를 출력할 수 있으므로 endl 은 결국 개행 을 하게 됩니다.

xendl 은 함수 모양으로 사용할 수 있지만 다음의 형태로도 사용 가능 합니다.

```
cout << xendl; // cout.operator<<(xendl) 즉, 함수 포인터를 인자로 받고 있습니다.
```

결국 "operator<<" 연산자 함수가 함수 포인터를 인자로 전달 받고 있기 때문에 가능합니다.

- **ostream 과 endl 의 연결**

ostream 클래스 안에는 모든 기본 타입에 대해서 "operator<<" 함수를 제공하고 있습니다. 또한, endl, hex 등의 함수를 전달 받기 위해서 함수 포인터를 인자로 받는 "operator<<" 함수도 제공하고 있습니다. 다음 코드는 ostream 과 xendl 의 관계를 보여주고 있습니다.

```
#include <stdio.h>

class ostream
{
public:
    ostream& operator<<(char   c) { printf("%c", c); return *this; }

    // char 뿐 아니라 모든 기본 타입에 대해서 "<<" 연산자를 재정의 합니다.
    // 또한 함수 포인터 버전도 제공합니다.
    ostream& operator<<(ostream&(*f)(ostream&) )
    {
        f(*this);    // f는 xendl을 가리키고 있습니다. 결국 xend(cout) 입니다.
        return *this;
    }
};
ostream cout;

ostream& xendl(ostream& os)
{
    os << '\n';
    return os;
}
int main()
{
    cout << xendl; // cout.operator<<(xendl) 입니다.
}
```

[참고] 코드를 단순화 하기 위해 std namespace 는 생략했습니다.

- **사용자 정의 조정자(iomanipulator) 만들기**

개행을 할 때 그냥 '₩n' 을 출력하면 될텐데, 왜 이렇게 복잡하게 설계 했을까요 ? endl 의 설계 의도중의 하나는 사용자가 endl 과 같은 도구를 직접 만들 수 있게 하는 것입니다. 다음 코드는 "탭" 만큼 띄워 쓰기를 할 수 있는 tab 이라는 조정자를 직접 만든 코드입니다.

```
ostream& tab(ostream& os)
{
    os << '\t';
    return os;
}
int main()
{
    cout << "A" << tab << "B" << endl;
}
```

이처럼 ostream 과 endl 의 원리를 정확히 이해 하면 재미 있는 도구를 만들 수도 있습니다.

- **실습과제**

다음 코드와 실행 결과를 참고 해서 화면에 메뉴를 출력하는 menu 를 만들어 보세요.

```
int main()
{
    cout << menu;
}
```

```
// 실행 결과
1. 짜장면
2. 김치찌게
3. 비빔밥
```

25 장. 증가 연산자 재정의

이번 시간에는 C/C++ 에서 가장 많이 사용되는 연산자 중의 하나인 증가(++)/감소(--) 연산자를 재정의하는 것에 대해서 살펴 보도록 하겠습니다. 감소(--)연산자는 증가 연산자와 거의 유사 하므로 증가 연산자만 만들도록 하겠습니다. 증가 연산자를 만드는 과정에서 연산자 재정의 뿐 아니라 많은 것을 배울 수 있습니다.

25.1 Integer

증가 연산자를 재정의 하기 전에 먼저 Integer 클래스를 하나 만들도록 하겠습니다. C 에서는 지역 변수가 자동으로 초기화 되지 않은 문제점이 있습니다. int 역할을 하는 Integer 라는 클래스를 하나 만들어서 초기값을 지정하지 않은 경우 자동으로 0 으로 되도록 만들어 보도록 하겠습니다.

```
class Integer
{
    int value;
public:
    Integer(int n = 0) : value(n) {}
    void print() { cout << value << endl; }
};
int main()
{
    int     n1; // 초기화 하지 않았습니다.
    Integer n2; // 객체이므로 생성자가 호출 되므로, 0으로 초기화 됩니다
    n2.print();
}
```

[참고] java 에는 표준 타입인 int 가 있고, 사용자 정의 타입인 Integer 가 있습니다.

int 대신 Integer 를 사용하려면 Integer 는 int 로 가능한 대부분의 연산을 제공해야 합니다.

++ 연산은 int 에서 가장 많이 사용되는 연산중의 하나 입니다. Integer 타입에 대해서 ++연산이 가능하도록 ++ 연산자를 만들어 보겠습니다.

25.2 일반 함수와 멤버함수 의 선택

연산자 재정의 함수는 일반 함수 또는 멤버 함수로 구현 할 수 있습니다. ++연산자의 경우도 멤버 또는 일반 함수로 구현 할 수 있습니다.

```
int main()
{
    Integer n(3);
    ++n;    // 1. n.operator++(), 멤버함수
            // 2. operator++(n)   멤버가 아닌 함수
}
```

++연산자의 경우 수행 후 상태가 변경됩니다. 함수안에서 객체의 상태를 변경하려면 일반 함수보다는 멤버함수가 편리하므로 멤버 함수로 구현하겠습니다. 물론, 멤버가 아닌 일반 함수로 구현해도 상관없습니다.

25.3 전위형(prefix) 증가연산자와 후위형 증가 연산자의 구별

증가 연산자의 경우 전위형과 후위형이 있습니다. 또한, 전위형과 후위형은 동작 방식이 다릅니다.

```
int main()
{
    int n = 3;
    int a = ++n; // n을 증가하고 증가된 n을 리턴 합니다.
    int b = n++; // n을 증가하고 증가되기 이전 값을 리턴 합니다.
}
```

그런데, 문제는 전위형과 후위형을 연산자 재정의 함수로 표현하면 결국 같은 모양이 됩니다. 그래서, C++에서는 후위형을 인자로 int 하나를 받도록 만들어야 합니다. 이때, int 타입 인자는 실제 사용되는 것은 아니고 전위형 과 구별하기 위한 약속입니다.

```
int main()
{
    Integer n;
```

```
    ++n;    // n.operator++() 함수를 찾게 됩니다.

    n++;    // n.operator++(int) 함수를 찾게 됩니다.
            // 인자인 int는 실제 사용되지 않고, 전위형과 구별하기 위한 약속입니다.
}
```

25.4 전위형(prefix) 증가연산자의 구현

전위형 증가 연산자는 자신을 먼저 증가하고, 증가된 자기 자신을 리턴 하게 됩니다. 따라서 다음 처럼 사용하는 것이 가능합니다.

```
int main()
{
    int n = 3;
    ++(++n);    // ++( 4로 증가된 n );
    cout << n << endl; // 5
}
```

따라서 전위형 증가 연산자를 만들 때는 자기 자신을 참조로 리턴 해야 합니다.

```
class Integer
{
    int value;
public:
    Integer(int n = 0) : value(n) {}
    void print() const { cout << value << endl; }

    Integer& operator++()   // 주의! 반드시 참조를 리턴 해야 합니다.
    {                       // 값을 리턴 하는 경우 임시객체가 생성 됩니다.
        ++value;
        return *this;
    }
};
int main()
{
    Integer n(3);
    ++(++n);
```

```
    n.print(); // 5
}
```

전위형 증가 연산자를 만들 때는 주의 할 점은 리턴 값을 참조로 해서 임시객체의 생성을 막아야 합니다. 위의 코드에서 전위형 증가 연산자의 리턴 값을 값 타입(Integer)로 변경한후 실행해 본 후 결과를 확인해 보세요.

25.5 후위형(prefix) 증가연산자의 구현

후위형 증가 연산자는 자신을 증가하지만 리턴 값은 증가 되기 이전 값을 리턴 합니다. 즉, 증가되기 이전 값을 보관하고 있다가 리턴 해 주어야 합니다. 다음 처럼 구현합니다.

```
// 후위형은 전위 형과 구별하기 위해 int 타입을 인자로 사용합니다.
// 실제 사용되지는 않습니다
Integer operator++(int)
{
    Integer temp = *this;   // 증가 되기전의 상태 보관
    ++value;
    return temp;    // 보관해둔 값을 리턴
}
```

결국 후위형은 지역 객체를 리턴 하게 되므로 참조를 리턴 하면 안됩니다.

- **정책을 한곳에 놓기**

Integer 클래스의 객체를 생성한 후에 ++ 연산자를 사용하면 내부 멤버 data 인 value 가 1 증가하게 됩니다. 그런데, 만약 Integer 클래스를 사용해서 홀수만 관리하기로 했다고 생각해 봅시다. 그렇다면 ++연산자에는 1 이 아닌 2 가 증가 되어야 합니다. 이제 코드는 몇 군데 수정이 되어야 할까요 ? 전위형과 후위형을 별도로 만들었으므로 2 곳을 변경해야 합니다. 이경우 후위형의 구현에서 전위형을 다시 호출하면 한곳에서 정책을 관리할 수 있습니다

```
Integer operator++(int)
{
    Integer temp = *this;
    ++(*this);  // value를 직접 증가하지 말고 전위형 증가 연산자를 다시 호출합니다.
```

```
    return temp;
}
```

[참고] 후위형 구현시 반드시 전위형을 사용해야 하는 것은 아닙니다. 직접 구현해도 문제는 없습니다.

25.6 완성된 코드

다음 코드는 전위형/후위형 증가연산자를 구현한 Integer 클래스 입니다.

```
class Integer
{
    int value;
public:
    Integer(int n = 0) : value(n) {}
    void print() const { cout << value << endl; }

    Integer& operator++()
    {
        ++value;// += 2;
        return *this;
    }
    Integer operator++(int )
    {
        Integer temp = *this;
        ++(*this);
        return temp;
    }
};
int main()
{
    Integer n(3);
    n++;
    n.print();
}
```

26 장. 함수 객체, 스마트 포인터, 대입연산자

26.1 함수 객체 (Function Object, Functor)

함수를 호출할 때 사용하는 "()" 도 재정의가 가능 합니다. "()" 연산자를 재정의 하면 객체를 마치 함수 처럼 사용할 수 있습니다.

```
class Plus
{
public:
    int operator()(int a, int b)
    {
        return a + b;
    }
};
int main()
{
    Plus p;
    int n = p(1, 2);// p는 객체이지만 함수 처럼 사용됩니다.
}
```

위 코드에서 p 는 Plus 타입의 객체입니다. 하지만 마치 함수처럼 "p(1,2)" 의 표현식으로 사용하고 있습니다. 원리는 다음과 같습니다.

```
a + b; // a.operator+(b)    // 결국 operator"연산자이름"() 형태 입니다.
a - b; // a.operator-(b)
a();   // a.operator()()    // 앞의 ()는 연산자 이름이고 뒤의 ()는 함수 호출
                            // 인자가 없는 경우 입니다.
a(1,2);// a.operator()(1,2) // 인자가 int 2개 있는 모양입니다.
```

결국, "p(1,2)" 는 컴파일러의 의해 "p.operator()(1,2)" 처럼 해석됩니다.

일반 함수를 사용하면 되는데, 왜 함수 객체를 사용할까요 ? 함수객체는 결국 클래스 이므로

- ✓ **멤버 데이터, 생성자, 소멸자 등 다양한 요소를 활용할 수 있고,**
- ✓ **특정한 상황에서는 함수 객체가 일반 함수 보다 빠른 경우가 있습니다.**

좀더 자세한 내용은 좀 어려운 주제이므로 "Advanced C++" 과정을 참고 하시기 바랍니다.

- **템플릿과 STL**

앞에서 만든 Plus 는 int 형 2 개의 덧셈만 할 수 있습니다. 템플릿을 사용하면 보다 재사용성을 높일 수 있습니다.

```
template<typename T> class Plus
{
public:
    T operator()(T a, T b) const
    {
        return a + b;
    }
};
int main()
{
    Plus<int> p;
    cout << p(1, 2) << endl;
}
```

또한, 이와 같은 함수 객체는 C++ 표준 라이브러리인 STL 에서 많이 사용됩니다. 그래서, 이미 C++표준에 plus 함수 객체가 이미 제공됩니다. plus 함수 객체를 사용하려면 <functional> 헤더를 포함 해야 합니다.

```
#include <iostream>
#include <functional> // c++표준 함수 객체를 사용하기 위한 헤더.
using namespace std;

int main()
{
    plus<int> p;
    cout << p(1, 2) << endl;
}
```

26.2 스마트 포인터 (Smart Pointer)

() 연산자를 재정의 하면 객체를 함수처럼 사용 할 수 있듯이, -> 연산자를 재정의 하면 객체를 포인터 처럼 사용할 수 있습니다. 다음 코드를 생각해 보세요.

핵심은 Ptr 클래스의 -> 연산자와 *연산자를 재정의 하는 코드 입니다.

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

class Car
{
    int color;
public:
    ~Car()    { cout << "Car 파괴" << endl; };
    void Go() { cout << "Car Go" << endl; }
};

class Ptr
{
    Car* obj;
public:
    Ptr(Car* p = 0) : obj(p) {}

    // 소멸자에서 자원을 삭제 합니다.
    ~Ptr() { delete obj; }

    // 아래 2줄이 스마트 포인터의 핵심 입니다.
    Car* operator->() { return obj; }
    Car& operator*() { return *obj; } // 주의 .. 반드시 참조를 리턴 해야 합니다.
};
int main()
{
    Ptr p = new Car; // Ptr p( new Car )

    p->Go();  // (p.operator->())Go()
              // (p.operator->())->Go()

    (*p).Go(); // p.operator*()
}
```

이와 같은 스마트 포인터의 가장 큰 장점은 진짜 포인터가 아닌 객체라는 점입니다. 객체는 생성자/복사생성자/소멸자/대입연산자 등을 만들수 있기 때문에, 생성/복사/대입/소멸의 모든 과정을 개발자가 제어할수 있게 됩니다.

- **템플릿과 복사 정책, shared_ptr**

앞에서 만든 Ptr 은 다음과 같은 문제점이 있습니다.

① **Car 의 포인터 역할만 합니다. 다른 타입도 사용가능 하게 하려면 템플릿으로 만들어야 합니다.**

② **디폴트 복사 생성자에의한 얕은 복사 현상이 있습니다. 반드시 복사 생성자를 제공해서 참조 계수 등의 방식으로 구현해야 합니다. (대입 연산자도 제공해야 합니다.)**

C++ 표준에는 shared_ptr 이라는 스마트 포인터를 제공하고 있습니다. 템플릿으로 되어 있고 참조계수 방식으로 복사 생성자를 제공하고 있습니다. 다음은 shared_ptr 를 사용하는 간단한 예제 입니다.

```
#include <iostream>
#include <memory>   // C++ 표준 스마트 포인터
using namespace std;

int main()
{
    shared_ptr<int> p1(new int);
    *p1 = 10; // int* 처럼 사용하면 됩니다.
    cout << *p1 << endl;

    shared_ptr<int> p2 = p1; // 이 순간 참조계수가 증가합니다.
    cout << p1.use_count() << endl; // 참조계수 출력
}
```

shared_ptr 를 사용하는 방법에 대한 보다 자세한 이야기는 "C++ STL Programming"과정을 참고 하시면 됩니다.

26.3 대입 연산자

다음의 멤버들은 사용자가 제공하지 않으면 컴파일러가 제공해 줍니다.

① **생성자를 제공하지 않으면 컴파일러는 디폴트 생성자를 제공합니다.**

② **소멸자를 제공하지 않으면 컴파일러는 소멸자를 제공합니다.**

③ **복사 생성자를 제공하지 않으면 컴파일러는 복사 생성자를 제공 합니다. (얕은 복사)**

또한, **사용자가 대입연산자를 제공하지 않으면 컴파일러는 대입연산자도 제공합니다.**

다음 코드를 생각해 봅시다.

```
class Point
{
public:
    int x = 0;
    int y = 0;
};
int main()
{
    Point p1;     // ok. 컴파일러가 제공한 디폴트 생성자를 사용합니다.
    Point p2(p1); // ok. 컴파일러가 제공한 복사 생성자를 사용합니다.

    Point p3;
    p3 = p1; // p3.operator=(p1) 이 필요합니다.
             // 사용자가 대입연산자를 제공하지 않으면 컴파일러가 제공해 줍니다.
}
```

지금 부터 대입연산자에 대해서 자세히 살펴 보도록 하겠습니다.

- **초기화와 대입**

객체를 만들 때 값을 넣는 것은 초기화 이고, 객체를 만든 후에 값을 넣는 것은 대입입니다. 각각 복사 생성자와 대입연산자가 사용됩니다.

```
Point p1;
Point p2 = p1; // 초기화 입니다. 복사 생성자가 호출됩니다.
```

```
Point p3;
p3 = p1; // 대입 입니다. 대입연산자가 호출됩니다.
```

- **대입연산자의 리턴값**

대입 연산자는 자기 자신을 리턴해야 합니다.

```
(a = b) = c;    // "a = b"는 a를 리턴합니다.
                // 최종적으로 a에는 c의 값이 대입됩니다.
```

따라서, 사용자 정의 타입에서 대입 연산자를 만들 때는 자기 자신을 참조로 리턴해야 합니다. 주의할 것은 값을 리턴하면 자신의 복사본이 임시객체로 생성되므로 반드시 값이 아닌 참조를 리턴해야 합니다.

```
Point& operator=(const Point& p)
{
    x = p.x;
    y = p.y;
    return *this;
}
```

- **컴파일러가 제공해주는 대입 연산자가 하는 일 - 얕은 복사**

사용자가 대입연산자를 제공하지 않으면 컴파일러가 제공해 줍니다. 컴파일러가 제공해 주는 대입연산자는 모든 멤버를 대입해 줍니다. 이 때, 클래스 내부에 포인터 멤버가 있다면 복사 생성자와 유사한 얕은 복사 현상이 발생합니다. 따라서, 사용자가 대입연산자를 직접 구현해서, 깊은 복사, 참조계수 또는 대입 금지 코드를 제공해야 합니다.

자세한 코드는 다음장의 String 만들기를 참고 하면 됩니다.

- **대입연산자는 반드시 멤버 함수로 구현해야 합니다.**

일반적으로 연산자 재정의 함수는 멤버 함수 또는 일반 함수로 구현 할 수 있습니다. 하지만 다음의 4 개의 연산자는 멤버 함수로만 구현 할 수 있습니다.

"=, (), [], ->"

27 장 String 클래스 만들기

C 언어 에서는 문자열을 처리하기 위해 char 배열 또는 char* 를 사용합니다. 그래서, 문자열을 대입이나 비교 등을 할 때 직관적인 연산자를 사용 할 수 없고 문자열 전용 함수를 사용해야 합니다. 하지만 C++의 경우 string 클래스를 사용하면 문자열을 직관적인 연산자를 사용해서 조작할 수 있습니다

이번 장에서는 지금 까지 배운 클래스 문법과 연산자 재정의 문법을 사용해서 C++의 표준 string 처럼 사용 할 수 있는 String 클래스를 만들어 보도록 하겠습니다.

앞으로 만들 String 클래스의 기본 요구 조건은 다음과 같습니다.

```
int main()
{
    String s1 = "hello";    // 1. 문자열 변수를 만들고
    cout << s1 << endl;     // 2. 화면 출력이 가능해야 합니다.

    String s2 = s1;         // 3. 문자열 복사 생성이 가능해야하고.
    cout << s2 << endl;

    String s3 = "world";
    s3 = s1;                // 4. 문자열 대입도 가능해야 합니다.
    cout << s3 << endl;

    s1 = s1;                // 5. 자신과의 대입도 가능해야 합니다.
    cout << s1 << endl;
}
```

- **생성자와 소멸자**

```
String s1 = "hello";    // 1. 문자열 변수를 만들고
```

s1 은 내부 적으로 “hello” 문자열을 보관하고 있어야 합니다. 결국, 생성자에서 동적 메모리 할당을 하고 소멸자에서 메모리를 해지 해야 합니다. 다음은 완성된 코드입니다.

```
class String
{
    char* buff;
```

```
    int   size;
public:
    String(const char* s)
    {
        size = strlen(s);
        buff = new char[size + 1];
        strcpy(buff, s);
    }
    ~String() { delete[] buff;  }
};
int main()
{
    String s1 = "hello";
}
```

- **화면 출력**

```
String s1 = "hello";    // 1. 문자열 변수를 만들고
cout << s1 << endl;     // 2. 화면 출력이 가능해야 합니다.
```

String 형 변수 s1 을 화면 출력하려면 "operator<<" 연산자를 제공해야 합니다. 또한, String 클래스 안에 friend 함수로 선언해야 합니다.

```
class String
{
    // ……
    friend ostream& operator<<(ostream& os, const String& s);
};
ostream& operator<<(ostream& os, const String& s)
{
    os << s.buff;
    return os;
}
```

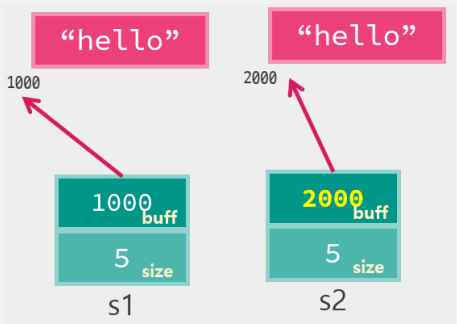
- **복사 생성자 - 깊은 복사**

```
String s2 = s1; // 복사 생성자가 사용됩니다.
```

String 클래스는 내부적으로 포인터 멤버가 가지고 있습니다. 클래스가 포인터를 가지고 있을 때 디폴트 복사 생성자는 얕은 복사 현상이 발생하므로, 반드시 사용자는 복사 생성자를 제공해야 합니다.

"깊은복사, 참조계수, 복사금지" 등 의 방법이 있습니다. 여기서는 "깊은 복사" 방법을 사용하겠습니다. 깊은 복사 로 구현한 복사 생성자 코드 입니다.

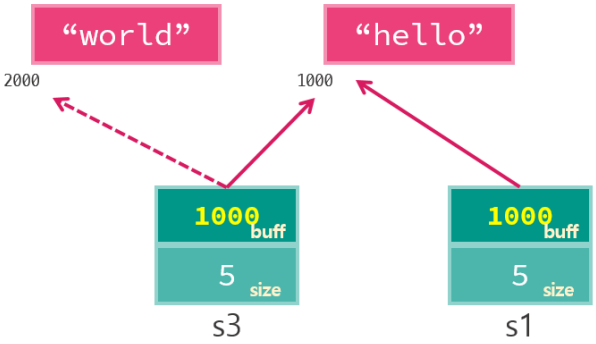
```
class String
{
    char* buff;
    int   size;
public:
    // ……
    // 깊은 복사로 구현한 복사 생성자
    String(const String& s)
            : size(s.size)
    {
        buff = new char[size + 1];
        strcpy(buff, s.buff);
    }
};
int main()
{
    String s1 = "hello";
    cout << s1 << endl;

    String s2 = s1; // 복사 생성자..
    cout << s2 << endl;
}
```



- **대입 연산자**

```
String s1 = "hello";
String s3 = "world";
s3 = s1;
```

사용자가 대입연산자를 제공하지 않으면 컴파일러가 제공합니다. 이때 컴파일러가 제공하는 대입 연산자는 모든 멤버를 대입하므로 얕은 복사 현상이 있습니다. 복사 생성자와 유사한 하지만 한가지 문제가 더 발생합니다.



디폴트 대입 연산자에 의해 얕은 복사가 발생한 후에는 "hello" 문자열 메모리가 2 번 delete 되는 문제점 외에도 "world" 문자열을 가진 메모리는 delete 되지 않은 문제가 발생합니다. 따라서, 대입연산자는

① **포인터가 아닌 size 멤버는 그냥 복사 합니다.**

② **자신이 사용하던 메모리("world" 문자열) 를 삭제 합니다. - 복사 생성자와 다른점 입니다.**

③ **s1 의 문자열("hello")을 깊은 복사 하기 위해 메모리를 할당하고 문자열을 복사 합니다.**

다음은 완성된 대입연산자 코드 입니다.

```
String& operator=(const String& s)
{
    size = s.size;

    // 자신의 버퍼를 지우고
    delete[] buff; // "world" 메모리 삭제..

   // 새로운 버퍼 할당
    buff = new char[size + 1];
    strcpy(buff, s.buff);

    return *this;
}
```

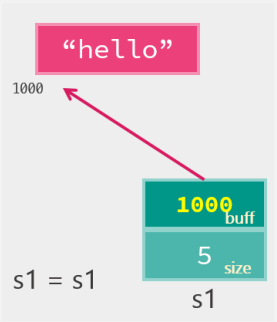
- **자신과의 대입을 조사**

대입 연산자 구현 관련해서 마지막으로 주의 해야 하는 점은 자기 자신과의 대입을 조사해야 한다는 점입니다. C 언어 에서는 변수에 자기 자신을 대입하는 것은 아무 문제가 없습니다.

```
s1 = s1;
```

그런데, 앞에서 만든 대입연산자의 경우 새로운 버퍼를 할당하기 전에 자신의 버퍼를 지우기 때문에 문제가 됩니다.

```
String& operator=(const String& s)
{
    size = s.size;

    // 이 순간에 문제가 발생합니다.
    delete[] buff;

    buff = new char[size + 1];
    strcpy(buff, s.buff);

    return *this;
}
```



해결책은 자신을 대입할때는 아무 일도 할 필요가 없으므로 대입 연산자의 첫번째 줄에 다음 코드를 추가 하면 됩니다.

```
String& operator=(const String& s)
{
    // 자신과의 대입을 비교한다.
    if (&s == this) return *this;
    // ……
    return *this;
}
```

- **완성된 코드**

다음은 완성된 코드입니다.

```
class String
{
    char* buff;
    int   size;
public:
    String(const char* s)
    {
        size = strlen(s);
        buff = new char[size + 1];
        strcpy(buff, s);
    }
    ~String() { delete[] buff;  }
    String(const String& s) : size(s.size)
    {
        buff = new char[size + 1];
        strcpy(buff, s.buff);
    }
    String& operator=(const String& s)
    {
        if (&s == this) return *this;

        size = s.size;
        delete[] buff;

        buff = new char[size + 1];
        strcpy(buff, s.buff);

        return *this;
    }
    friend ostream& operator<<(ostream& os, const String& s);
};
ostream& operator<<(ostream& os, const String& s)
{
    os << s.buff;
    return os;
}

int main()
{
    String s1 = "hello";
    cout << s1 << endl;

    String s2 = s1;
```

```
    cout << s2 << endl;

    String s3 = "world";
    s3 = s1;
    cout << s3 << endl;

    s1 = s1;
    cout << s1 << endl;
}
```

- **실습문제**

앞에서 만든 String 클래스를 참조 계수 기반으로 만들어 보세요

SECTION 5

Standard Template Library

STL 은 1998 년 발표된 C++표준 라이브러리 로서 다양한 자료구조와 컨테이너를 제공하고 있습니다.

대부분의 요소가 템플릿으로 되어 있고, 자료구조와 알고리즘을 분리해서 재사용성이 뛰어난 라이브러리 입니다.

이번장에서는 STL 의 핵심 요소에 대해서 살펴 보도록 하겠습니다.

이번 장에서 다음과 같은 개념을 배우게 됩니다.

"STL 소개, 컨테이너(container), 반복자(iterator), 알고리즘(algorithm)"

28 장. STL 소개

28.1 STL 소개

1979 년 C++이 처음 탄생한 이후, 표준화의 필요성을 깨달은 C++ 진영은 1990 년 C++ 표준 위원회를 설립합니다. 그리고, 1998 년 드디어 C++ 1 차 표준안을 발표합니다. 이때, 발표된 표준안에는 표준 문법 뿐 아니라 표준 라이브러리도 같이 발표 하게 됩니다.

표준 라이브러리에 있는 대부분의 요소가 "template"으로 제공되기 대문에 C++ 표준라이브러리를 "STL (Standard Template Library)" 라고 부릅니다. C++표준 라이브러리의 설계 목표 중 하나는 특정 분야 만을 위한 라이브러리가 아닌 다양한 분야의 S/W 개발에 도움을 줄 수 있도록 범용적인 라이브러리를 제공하는 것입니다.

1998 년 발표된 최초의 STL 은 대부분 stack, queue, list, tree 등의 자료구조를 나타내는 클래스와 선형검색, 이진 검색, 정렬, 순열등의 다양한 알고리즘을 제공하는 함수로 구성되어 있었습니다. 하지만, C++11 부터는 멀티 스레드, 정규 표현식, 날짜 와 시간, 스마트 포인터 등의 요소가 추가 되었고, 앞으로 네트워크, file system, web service, gui 등을 추가로 제공할 예정입니다.

28.2 STL 맛보기

C 언어에서는 문자열을 다루기 위해 char 배열 또는 char* 를 사용 합니다. 또한, 문자열을 복사, 비교 등을 하려면 =, == 등의 연산자가 아닌 strcpy, strcmp 등의 함수를 사용해야 합니다. 하지만, C++은 string 클래스를 제공하기 때문에 문자열 관련 연산을 수행할 때 비교적 직관적으로 코드를 작성 할 수 있습니다. 또한, STL 에는 복소수를 다루기 위해서 complex 클래스도 제공합니다.

그리고, 이런 타입들을 보다 쉽고 직관적으로 사용할 수 있게 하기 위해 +, - 등의 연산자를 재정의해서 제공하고 있습니다.

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

int main()
{
    string s1 = "hello";
    string s2 = s1 + s1; // + 연산자가 재정의 되어 있습니다.
    cout << s2 << endl;

    char c = s2[3]; // [] 연산자도 재정의 되어 있습니다.
```

```
    const char* s = s2.c_str(); // string을 char* 로 변형하려면
                                // c_str() 멤버 함수를 호출하면 됩니다.
    complex<int> c1(1, 1);
    complex<int> c2(2, 2);
    complex<int> c3 = c1 + c2;
    cout << c3 << endl;
}
```

결국 문자열 처리를 위해 char* 나 char 배열 대신 string 을 사용하면 훨씬 간결하게 프로그램을 작성할 수 있게 됩니다.

28.3 STL 의 핵심 개념

C++11 이 나오기 전에는 STL 에는 자료구조와 알고리즘위주의 라이브러리를 제공 했습니다. C++11 이 등장하면서 멀티 스레드, 스마트 포인터, 정규 표현식, 날짜와 시간 등의 다양한 분야의 라이브러리가 추가 되었습니다.

이번 장에서는 STL 의 다양한 요소 중 "자료구조와 알고리즘" 요소에 대해서 살펴 보도록 하겠습니다.

STL 에는 Data 를 저장하고 조작하기 위해 아래의 3 가지 요소를 제공합니다.

① **컨테이너(Container) : Data 를 저장하기 위한 자료구조**

② **반복자(Iterator) : 컨테이너의 요소를 열거하기 위한 포인터와 유사하게 동작하는 객체**

③ **알고리즘(Algorithm) : 컨테이너에 있는 요소에 대한 검색, 정렬 등을 수행하기 위한 일반 함수**

하나씩 차례대로 살펴 보도록 하겠습니다.

29 장. 컨테이너(Container)

29.1 컨테이너 개념

일반적으로 프로그램을 개발 할 때는 stack, queue, list 등의 자료구조가 많이 필요 합니다. STL 에는 다음과 같은 다양한 자료구조를 클래스로 제공하고 있습니다. STL 에서는 이와 같은 자료구조를 보통 "컨테이너(Container)"라 고 합니다. STL 이 제공하는 컨테이너에는 다음과 같은 것들이 있습니다.

클래스이름	헤더파일	자료구조
array vector	<array> <vector>	C 의 배열과 유사한 형태의 자료구조 입니다. 연속된 메모리를 사용하는 자료구조 입니다.
list, forward_list	<list> <forward_list>	더블리스트와 싱글 리스트 입니다.
Deque	<deque>	list 와 vector 의 혼합형 자료구조 입니다.
Set Map	<set> <map>	tree 기반의 자료구조 입니다
unordered_set unordered_map	<unordered_set> <unordered_map>	hash 기반의 자료구조 입니다.
stack queue prioriry_queue	<stack> <queue> <queue>	스택 큐 우선순위 큐

이 외에도 tuple, valarray 등의 자료구조가 있습니다.

29.2 컨테이너의 특징

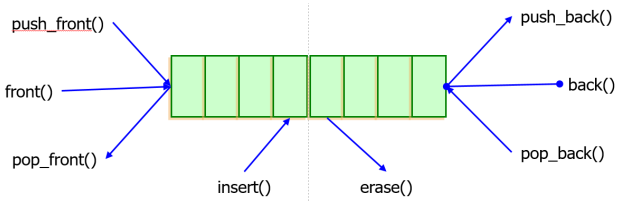
STL 의 컨테이너에는 몇가지 특징이 있습니다.

- **템플릿으로 만들어져 있습니다.**

STL(Standard Template Library)은 이름이 의미 하듯 대부분의 요소가 템플릿으로 되어 있습니다. 따라서, 다양한 타입을 사용할 수 있습니다.

- **멤버 함수의 이름이 동일 합니다.**

대부분의 컨테이너에서 앞에 요소를 삽입하려면 push_front, 중간에 삽입하려면 insert, 뒤에 삽입하려면 push_back 함수를 사용합니다.



멤버 함수의 이름이 동일 하기 때문에 소스 코드의 변경 없이 컨테이너만 변경해 가면서 성능을 테스트해 볼 수 있습니다.

```
#include <iostream>
#include <list>
#include <vector>
#include <deque>
using namespace std;

int main()
{
    // 컨테이너의 이름만 변경해서 테스트 해 볼 수 있습니다.
    //list<int>  c;
    //deque<int> c;
    vector<int>  c;

    for (int i = 0; i < 10; i++)
        c.push_back(i);
```

```
    for (auto& n : c)
        cout << n << endl;
}
```

차이가 있다면 vector 의 경우는 push_front 함수가 없는데, 그 이유는 연속된 메모리인 vector 의 앞에 요소를 삽입할 경우에는 항상 메모리 복제가 발생하므로 성능 저하가 있게 됩니다. 그래서, 컨테이너의 앞쪽에 요소를 삽입하는 작업이 필요하다면 다른 컨테이너를 사용하라는 의도를 가지고 push_front 를 제공하지 않는 것입니다.

- **제거와 리턴을 동시에 하지 않습니다.**

stack 에서 값을 넣을 때는 push, 값을 꺼낼 때는 pop 함수를 사용합니다. 하지만, STL 의 stack 은 pop 함수가 제거만 하고 리턴을 하지 않습니다. 값을 꺼내려면 top 함수를 사용해야 합니다.

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

int main()
{
    stack<int> s;
    s.push(10);
    s.push(20);

    //int ret = s.pop();// error. pop()은 제거만 하고 리턴하지 않습니다.
    int ret = s.top();  // ok. 20을 꺼냅니다. 하지만 제거되지 않습니다.
    ret = s.top();      // 계속 20 입니다.
    s.pop();            // 20을 제거 합니다.
    ret = s.top();      // 이제 10이 나옵니다.
}
```

stack 뿐 아니라 list, vector 등에서도 pop_back 함수는 제거만 하고 리턴 받으려면 back 함수를 사용해야 합니다. 물론, back 함수는 꺼내기만 하고 제거하지는 않습니다.

```
int ret = 0;
list<int> s = { 1,2,3 };
```

```
ret = s.pop_back(); // error. pop_back()은 제거만 하고 리턴하지 않습니다.
ret = s.back();     // ok. 3을 꺼내지만 제거 되지는 않습니다.
```

30 장. 반복자(iterator)

30.1 반복자(Iterator) 개념

STL 의 반복자는 포인터와 유사한 객체로 컨테이너의 요소에 접근할 때 사용합니다. 배열의 요소를 열거할 때 포인터를 사용할 수 있듯이 컨테이너의 모든 요소를 열거할 때 반복자를 사용 할 수 있습니다. 컨테이너의 begin() 함수를 통해서 첫번째 요소를 가리키는 반복자를 얻을 수 있습니다.

```
int x[10] = { 1,2,3,4,5,6,7,8,9,10 };       // 배열
vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };   // STL vector

int* p1 = x;            // 배열의 첫번째 요소를 가리키는 포인터 입니다.
auto p2 = v.begin();    // vector의 첫번째 요소를 가리키는 반복자입니다.
++p2;                   // 포인터와 유사하게 ++로 이동하고
cout << *p2 << endl;    // * 연산자로 값을 꺼낼 수 있습니다.
```

[참고] 물론, 배열이나 vector 는 모두 연속된 메모리 이므로 배열연산자([]) 를 통해서도 각요소에 접근 할 수 있습니다. 하지만, list 등은 [] 연산이 불가능하므로 반복자를 통해서 요소에 접근해야 합니다.

위 코드에서 반복자 p2 는 포인터와 거의 유사하므로 ++, * 등을 통해서 요소에 접근할 수 있습니다.

30.2 반복자 (iterator) 의 장점

C 에서 배열 등의 연속된 메모리는 시작 주소를 구한 후 ++연산자를 통해서 다음 요소로 이동할 수 있습니다. 하지만, linked list 는 연속된 메모리 공간이 아니므로 다음으로 이동하기 위해서는 next 요소를 참조해서 이동해야 합니다.

하지만, STL 의 반복자는 vector 의 반복자나 list 의 반복자나 모두 ++연산을 하면 다음 요소로 이동할 수 있습니다.

반복자의 원리는 ++연산자를 재정의해서 list 의 경우 next 를 참조 해서 이동하도록 만들어져 있습니다.

30.3 반복자 타입

반복자(iterator)는 컨테이너에 내포되어 있는 iterator 라는 타입 입니다. 따라서, v 가 vector<int> 타입일때 첫번째 요소를 가리키는 반복자를 꺼내려면 다음 처럼 합니다.

```
vector<int>::iterator p = v.begin();
```

반복자 타입이 좀 복잡해 보인다면, C++11 의 auto 를 사용하면 간단하게 코드를 간단하게 만들 수 있습니다

```
auto p = v.begin();
```

30.4 반복자 꺼내기 - begin(), end()

컨테이너의 멤버 함수인 begin, end 함수를 사용하면 반복자를 얻을 수 있습니다.

begin()	컨테이너의 첫번째 요소를 가리 키는 반복자를 리턴 합니다.
end()	컨테이너의 **마지막 요소 다음을** 가리 키는 반복자를 리턴 합니다. "past the end" 라고 합니다.

[참고] "마지막 요소의 다음"이라는 용어가 이상해 보일 수 있는데, linked list 같은 경우 마지막 Node 의 next 항목에는 '0'등을 넣는 경우가 있습니다. 이경우, 마지막 요소 다음이라는 의미는 0 번지를 가리키는 반복자가 됩니다.

주의 할 점은 end 함수가 리턴 하는 반복자는 컨테이너의 마지막 요소가 아닌 마지막 다음을 가리키므로 역 참조 등의 연산을 사용하면 안됩니다.

```
int main()
{
    vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };

    auto p1 = v.begin();
    *p1 = 10; // ok. 첫번째 요소 1을 10으로 변경합니다.

    auto p2 = v.end();
    *p2 = 10; // runtime error. p2는 마지막 요소가 아닌 다음을 가르킵니다.
```

```
}
```

v.end()는 반복자를 가지고 컨테이너의 요소를 이동 할 때 끝에 도달 했는 지를 파악하는 용도로 만 사용해야 합니다. 다음 코드는 반복자를 사용해서 vector 의 모든 요소를 출력하는 코드 입니다.

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

int main()
{
    vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };
    auto p1 = v.begin();

    // p1이 vector의 끝에 도달 했는지를 파악하기 위해 v.end()와 비교 합니다.
    while( p1 != v.end() )
    {
        cout << *p1 << endl;
        ++p1;
    }
}
```

반복자를 꺼낼 때 사용하는 begin, end 함수는 멤버 함수 뿐 아니라 일반 함수 로도 제공 됩니다. 멤버가 아닌 일반 함수 begin 은 STL 컨테이너 뿐 아니라 일반 배열에도 사용할 수 있습니다

```
int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };
auto p1 = v.begin();
auto p2 = begin(v); // 위 코드와 완전히 동일 합니다.

int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
auto p3 = x.begin(); // error. x는 객체가 아니므로 멤버 함수가 있을 수 없습니다.
auto p4 = begin(x);  // ok.    p4는 x 배열의 첫번째 요소의 주소 입니다.
```

멤버 함수 begin 대신 일반 함수 begin 을 사용하면 컨테이너 사용하다가 일반 배열로 변경하는 코드를 쉽게 작성할 수 있습니다.

```
//vector<int> v = { 1,2,3,4,5,6,7,8,9,10 }; // STL vector
int v[10] = { 1,2,3,4,5,6,7,8,9,10 };           // 배열

auto p = begin(v); // v는 STL 컨테이너 뿐 아니라 배열 이어도 문제 없습니다.
```

30. 5 컨테이너의 요소를 열거하는 3 가지 방법

STL 의 컨테이너안에 있는 요소를 열거하는 방법은 크게 3 가지 정도를 사용할 수 있습니다.

① 배열 연산자([])를 사용하는 방법 – vector, array, deque 등의 컨테이너에서 사용 할 수 있습니다. list 등에서는 사용할 수 없습니다.

② ranged-for 문을 사용하는 방법 – 대부분의 컨테이너에서 사용할 수 있습니다.

③ 반복자(iterator)를 사용해서 열거하는 방법 – 대부분의 컨테이너에서 사용할 수 있습니다.

다은 코드는 3 가지 방법에 대한 간단한 예제 입니다.

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

int main()
{
    vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };

    // 방법 1. [] 연산자 사용
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << endl;

    // 방법 2. ranged-for
    for (auto& n : v)
        cout << n << endl;

    // 방법 3. iterator 사용
    auto p = begin(v);
    while (p != v.end())
    {
```

```
        cout << *p << endl;
        ++p;
    }
}
```

어떤 방법이 좋은 지는 상황에 따라 다를 수 있습니다. 단순하게 요소에 대한 화면 출력이라면 ranged-for 정도가 간단하고 무난한 방법 입니다.

31 장. 알고리즘(algorithm)

31.1 알고리즘(algorithm) 함수 개념

STL 에는 컨테이너에 저장된 data 에 다양한 연산을 수행할 수 있는 알고리즘 함수를 제공합니다. 이번에는 알고리즘 함수에 대해서 자세히 살펴 보겠습니다.

- **알고리즘 함수는 멤버함수가 아니다**

list 안에 있는 요소에서 20 을 찾는 find 함수를 생각해 봅시다. 만약 find 함수가 멤버 함수로 제공된다면 다음과 같은 모양이 될 겁니다.

```
s.find(20);
```

이번에는 vector 에서도 20 을 찾고 싶다고 생각해 봅시다. 동일한 원리라면 아래 처 럼 될 것입니다.

```
s.find(20);  // list<int>   에서 값을 찾는 코드
v.find(20);  // vector<int> 에서 값을 찾는 코드
```

그런데, 이처럼 find 를 멤버 함수로 만들게 된다면 list, vector, string, deque 등 모든 컨테이너에 전부 만들어야 합니다. 하지만, find 를 만드는 방법은 list 건 vector 건 다르지 않습니다. 그렇다면, 멤버 함수로 만들지 말고 일반 함수로 만들면 어떨까요 ?

그런데, 문제는 find 를 만드는 방법(알고리즘)은 동일 하지만, 다음으로 이동하는 방식은 연속된 메모리인 vector 와 연속된 메모리가 아닌 list 가 서로 다르다는 점입니다.

하지만, STL 의 반복자를 사용하면 list 와 vector 에서 모두 동일한 방법인 ++를 사용해서 다음으로 이동 할 수 있습니다.

그래서, STL 은 find 와 같은 함수를 멤버 함수로 제공하지 않고 일반 함수로 제공합니다. 한 개의 find()함수로 모든 컨테이너에 사용할 수 있다는 장점을 가지게 됩니다.

[참고] find 는 결국 함수 템플릿 이므로 진짜 함수가 한 개인 것은 아닙니다. list 와 vector 에 대해 find 를 사용하면 list 용 find 와 vector 용 find 함수가 각각 코드 생성됩니다. 하지만, 라이브러리 설계자 입장에서는 find 를 템플릿으로 하나만 제공하면 되므로 분명한 장점이 있습니다..

이때, find 같은 함수를 STL 에서는 "알고리즘(algorithm)" 이라고 부르고 <algorithm> 헤더를 포함해야 합니다.

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    list<int>   s = { 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };

    // find 알고리즘 하나로 list와 vector에서 모두 값을 검색합니다.
    auto p1 = find(s.begin(), s.end(), 20);
    auto p2 = find(v.begin(), v.end(), 20);

    // reverse 알고리즘 하나로 list와 vector 를 모두 뒤집습니다.
    reverse(s.begin(), s.end());
    reverse(v.begin(), v.end());
}
```

결국 STL 의 핵심 설계 철학은 다양한 알고리즘 함수를 멤버 함수가 아닌 일반 함수로 제공해서 하나의 함수(템플릿)으로 다양한 컨테이너에서 사용할 수 있도록 만들었다는 점입니다.

STL 에는 "정렬, 선형검색, 이진검색, 순열"등의 다양한 연산을 제공하는 알고리즘 함수를 제공합니다.

31.2 알고리즘의 정책 변경과 람다 표현식 (Lambda Expression)

STL 의 다양한 알고리즘을 사용할 때 해당 알고리즘이 사용하는 정책을 변경하고 싶을 때가 있습니다. vector 안에 있는 요소를 정렬하는 코드를 생각해 봅시다. 아래 코드를 실행하면 다음과 같은 결과를 얻을 수 있습니다.

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

int main()
{
    vector<int> v = { 1,3,5,7,9,2,4,6,8,10 };

    sort(v.begin(), v.end());

    for (auto& n : v)
        cout << n << " ";
}
```

```
// 실행결과
1 2 3 4 5 6 7 8 9 10
```

sort 알고리즘은 기본적으로 less 연산자("<") 를 사용해서 2 개의 값을 비교하기 때문에 오름차순으로 정렬이 됩니다. 내림 차순으로 정렬을 하고 싶으면 사용자가 비교 함수를 만들어서 sort 의 3 번째 인자로 전달하면 됩니다.

```
bool cmp_greater(int a, int b)
{
    return a > b;
}
int main()
{
    vector<int> v = { 1,3,5,7,9,2,4,6,8,10 };
    sort(v.begin(), v.end(), cmp_greater); // 마지막 인자로 비교 함수를 전달합니다.
}
```

위코드는 결국 내림 차순으로 정렬 하므로 10, 9, 8, …, 1 로 정렬 됩니다.

결국 sort 알고리즘 함수가 사용하는 정책을 변경하려면 아래 처럼 sort 의 3 번째 인자로 "정렬을 위해 2 개의 요소를 비교" 하는 정책을 보내면 됩니다.

```
sort(v.begin(), v.end(), "정렬을 위해 두개의 요소를 비교하는 비교정책" );
```

이때 비교 정책으로 사용가능한 것은 아래의 3 가지가 있습니다.

① 일반 함수

② 함수 객체 – 사용자가 만들거나 <functional> 헤더안에 있는 것들

③ 람다 표현식(lambda expression) – C++11 부터 제공.

다음 코드에서 위의 3 가지 방법을 사용하는 예제를 볼 수 있습니다.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

bool cmp_greater(int a, int b)
{
    return a > b;
}

int main()
{
    vector<int> v = { 1,3,5,7,9,2,4,6,8,10 };

    // 1. 비교정책으로 일반 함수를 사용하는 경우
    sort(v.begin(), v.end(), cmp_greater);

    // 2. 비교정책으로 STL이 제공하는 함수 객체를 사용하는 경우
    greater<int> g;
    sort(v.begin(), v.end(), g);

    // 3. 비교정책으로 람다표현식을 사용하는 경우
    sort(v.begin(), v.end(), [](int a, int b) { return a > b; });
}
```

위 코드에서 main 함수의 마지막 줄에서 sort 함수의 3 번째 인자로 전달되는 아래 코드를 람다 표현식(Lambda Expression) 이라고 합니다.

```
[](int a, int b) { return a > b; }
```

첫번째 요소이 "[]"는 람다 표현식이 시작됨을 알리는 "Lambda Introducer"라고 합니다. Lambda Introducer 다음에는 함수와 동일한 표현식을 적게 됩니다.

람다 표현식은 다른 언어에 있는 "클로져(Closure)" 문법과 유사합니다. 람다 표현식의 정확한 원리는 좀 복잡 하므로 "Advanced C++(C++11/14)" 과정에서 다루게 됩니다.

31.3 C++11 과 STL

C++11 의 등장과 함께 STL 에도 많은 요소가 추가 되었습니다. 기존의 STL 이 "자료구조와 알고리즘" 위주 였다면 C++11 에서는 "멀티 스레드, 동기화, 스마트 포인터, 정규표현식, 시간, function 과 bind" 등 많은 요소가 추가되었습니다.

이와 같은 주제에 대한 자세한 설명은 "C++ STL Programming" 과정을 참고 하시기 바랍니다.

SECTION 6

Exception & Stream

이번 장에서는 객체지향 언어에서 사용하는 에러 처리 방식인 예외 에 대해서 간략히 살펴 봅니다.

그리고, 마지막으로 파일 입출력에 대해서 간단히 살펴 봅니다.

32 장. 예외 처리(exception)

32.1 C 언어에서의 에러 처리

C 언어에서는 함수가 실패를 할 경우 반환값을 통해서 호출자에게 알려주게 됩니다.

```
int foo()
{
    if (실패)
        return -1; // 실패한 경우. 약속된 값을 반환 합니다.

    return ...;
}
```

이와 같은 방식에는 몇 가지 문제가 있습니다.

① 반환된 값이 실패를 나타내는지 연산의 결과 인지 혼란스러울 때가 있습니다. (위 코드에서 -1 이 연산의 결과 인지, 실패를 나타내는지 호출자는 알수가 없습니다.)

② 함수가 에러를 반환 할 때, 호출자 에게 반드시 에러를 처리하라고 강제로 지시할 방법이 없습니다.

```
int main()
{
    foo();  // foo 가 실패를 반환 할수 있지만 호출자는
            // 반드시실패를 조사해야 하는 것은 아닙니다.
}
```

32.2 예외를 사용한 에러 처리

C++을 비롯한 객체지향 언어에서는 반환값과 에러를 분리하기 위해 예외 처리 기법을 사용합니다.

- **예외의 전달**

먼저 함수에서 호출자에게 실패를 알려주기 위해서 throw 를 사용합니다.

```
int foo()
{
```

```
    if (실패)
        throw 1;    // 실패한 경우, 예외를 throw 합니다.
                    // 리턴값과 실패의 전달이 분리됩니다.
    return 0;
}
int main()
{
    foo();  // foo가 예외를 던질때, 호출자는 반드시 예외를 처리해야 합니다.
            // 예외를 처리하지 않은 경우. 프로세스는 종료 됩니다.
}
```

함수가 예외를 던지면 호출자는 반드시 예외를 처리해야 합니다. 예외 처리를 하지 않은 경우, 프로그램은 비정상 종료를 하게 됩니다.

- **예외의 처리**

함수가 예외를 던지면 호출자는 try 블록을 설치 하고 catch 를 사용해서 예외를 잡아서 처리해야 합니다.

```
int main()
{
    // 예외 가능성이 있는 함수를 호출할 때는 try 블럭 안에 넣습니다.
    try
    {
        foo();
    }
    catch (int n)   // int 형 예외를 catch 합니다.
    {
    }
    catch (...)     // ... 은 모든 종류의 예외를 catch 할수 있습니다.
    {
    }
    // 예외를 catch 한 경우 프로그램은 계속 실행될수 있습니다.
}
```

예외를 잡아서 처리한 경우 프로그램은 계속 정상 실행될수 있습니다.

32.3 예외 클래스

함수에서 예외를 던질 때 예외 전용 클래스를 사용해서 전달하는 것이 원칙입니다. 또한, C++ 표준에는 exception 이라는 이름의 예외 클래스를 제공하는데, 사용자 정의 예외를 만들때는 보통 이 클래스로부터 상속 받은후 what() 가상함수를 재정의 하는 것이 원칙입니다.

아래 코드에서 예외 전용 클래스를 만들어 사용하는 것을 볼수 있습니다.

```
class MemoryException : public exception
{
public:
    virtual char const* what() const
    {
        return "memory exception";
    }
};
int foo()
{
    if ( 1 )    // 실패
        throw MemoryException();

    return 0;
}
int main()
{
    try
    {
        foo();
    }
    catch (const MemoryException& m)
    {
        cout << m.what() << endl;
    }
}
```

32.4 C++ 표준 예외 클래스

- sd::bad_alloc

new 연산자로 메모리를 할당 할 때 메모리가 부족한 경우 std::bad_alloc 예외가 던져 집니다.

```
int main()
{
    int* p = nullptr;

    try
    {
        p = new int[1000];
    }
    catch (std::bad_alloc& e)
    {
        //......
    }
}
```

STL 라이브러리는 예외를 많이 사용하지는 않지만 일부 멤버 함수가 예외를 던지는 경우가 있습니다.

```
int main()
{
    vector<int> v = { 1,2,3 };
    try
    {
        v.at(5) = 5; // 잘못된 index를 사용했습니다.
    }
    catch (std::out_of_range& e)
    {
    }
}
```

32.5 noexcept

함수가 리턴값이 없음을 나타낼때는 void 를 표기합니다. 유사하게, 함수가 예외가 없음을 나타낼때는 함수 () 뒤에 noexcept 를 표기합니다.

```
void foo();             // 이 함수는 예외가 있을수도 있고, 없을수도 있습니다
void goo() noexcept;    // 이 함수는 예외가 없습니다.
```

noexcept 를 사용 했을때의 장점등, 예외에 대한 보다 깊이 있는 예외에 대한 이야기는 “C++ Intermediate” 과정을 참고하시기 바랍니다.

33 장. Stream

33.1 <iostream>

앞에서 배웠듯이 cout 과 cin 은 객체이므로 <<, >> 연산자 이외에도 몇가지 멤버 함수를 제공하고 있습니다.

▪ cin 과 멤버 함수

cin 으로 입력을 받을 때 입력 성공/실패 여부를 확인하려면 fail() 멤버 함수를 사용하면 됩니다.

```
int main()
{
    int n = 0;
    cin >> n;   // 정수를 입력 받으려고 합니다. 'a' 를 입력해 보세요
                // a는 정수가 아니고 문자 이므로 입력에 실패 합니다.
    if (cin.fail())
        cout << "입력 실패" << endl;
}
```

또한, cin 이 입력에 실패 한 경우, 다시 입력 받으려면

① clear() 멤버함수를 호출해서 cin 의 state 를 reset 한후,

② ignore() 멤버 함수를 사용해서 입력 버퍼를 비운뒤에 다시 입력을 받아야 합니다.

```
int main()
{
    int n = 0;
    while (1)
    {
        cin >> n;
        if (cin.fail())
        {
            cin.clear();            // cin 의 state 를 clear 합니다.
            cin.ignore(100, '\n');  // 입력 버퍼를 비웁니다.
            continue;               // 다시 입력 받습니다.
        }
        break;
    }
    cout << "입력 성공 : " << n << endl;
```

```
}
```

▪ 단어 입력 vs 문장 입력

cin 과 >> 연산자를 사용하면 단어를 입력 받을 수 있습니다. cin 을 사용해서 단어가 아닌 문장을 입력 받으려면 getline 멤버 함수를 사용합니다.

```
int main()
{
    string s;
    cin >> s;           // "aaa bbb" 를 입력해 보세요
    cout << s << endl;  // "aaa" 만 출력 됩니다.
}
```

getline 함수를 사용하면 단어가 아닌 문장 전체를 입력 받을수 있습니다.

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

int main()
{
    string s;
    getline(cin, s);    // "aaa bbb" 를 입력해 보세요
    cout << s << endl;  // "aaa bbb" 가 출력 됩니다.
}
```

33.2 파일 입출력

C++ 표준은 표준 입출력 뿐 아니라 파일로의 입출력을 위한 file stream 도 제공하고 있습니다. 파일 스트림을 사용하려면 <fstream> 헤더 파일을 포함하면 됩니다.

▪ 출력 파일 스트림 – ofstream

파일에 출력 하려면 ofstream 클래스를 사용합니다. ofstream 클래스는 cout 의 타입인 ostream 클래스로부터 파생 되었기 때문에 ostream 과 사용법이 거의 유사합니다.

다음과 같은 방식으로 파일에 출력 할 수 있습니다.

```
#include <iostream>
#include <fstream>  // 파일 스트림.
using namespace std;

int main()
{
    ofstream f("a.txt");    // 출력 파일 a.txt를 생성합니다.
    cout << "hello";    // 표준 출력으로 "hello"를 출력합니다.
    f    << "hello";    // 파일 a.txt에 "hello"를 출력합니다.
}
```

▪ 입력 파일 스트림 – ifstream

파일로부터 입력 받으르면 ifstream 클래스를 사용 합니다. Ifstream 클래스는 cin 의 타입인 istream 으로부터 파생 되었기 때문에 istream 클래스와 사용법이 거의 유사합니다.

아래 코드는 파일로 부터 한줄씩 입력을 받아서 표준 출력으로 출력하는 코드 입니다.

```
#include <iostream>
#include <fstream>  // 파일 스트림.
using namespace std;

int main()
{
    ifstream f("a.txt");    // 출력 파일 a.txt를 생성합니다.

    string s;
    while (getline(f, s))       // 파일에서 한줄을 입력 받습니다.
    {                       //파일 끝에 도달하면 반복문을 탈출합니다.
```

```
        cout << s << endl;
    }
}
```

- **fstream – 입출력 파일**

ofstream 은 파일로 출력, ifstream 은 파일로부터 입력 할 때 사용합니다. 입출력을 동시에 하려면 fstream 클래스를 사용하면 됩니다.

```
#include <iostream>
#include <fstream>  // 파일 스트림.
#include <string>
using namespace std;

int main()
{
    fstream f("a.txt", ios_base::in | ios_base::out | ios_base::trunc);

    if (f.is_open())
    {
        f << "hello";

        f.seekp(0);

        string s;
        f >> s;

        cout << s << endl;
    }
}
```

파일 입출력 관련 다양한 작업은 fstream 클래스의 멤버 함수를 사용하면 됩니다.

cppreference.com 에서 fstream 클래스의 멤버 함수를 참고 하시면 됩니다.