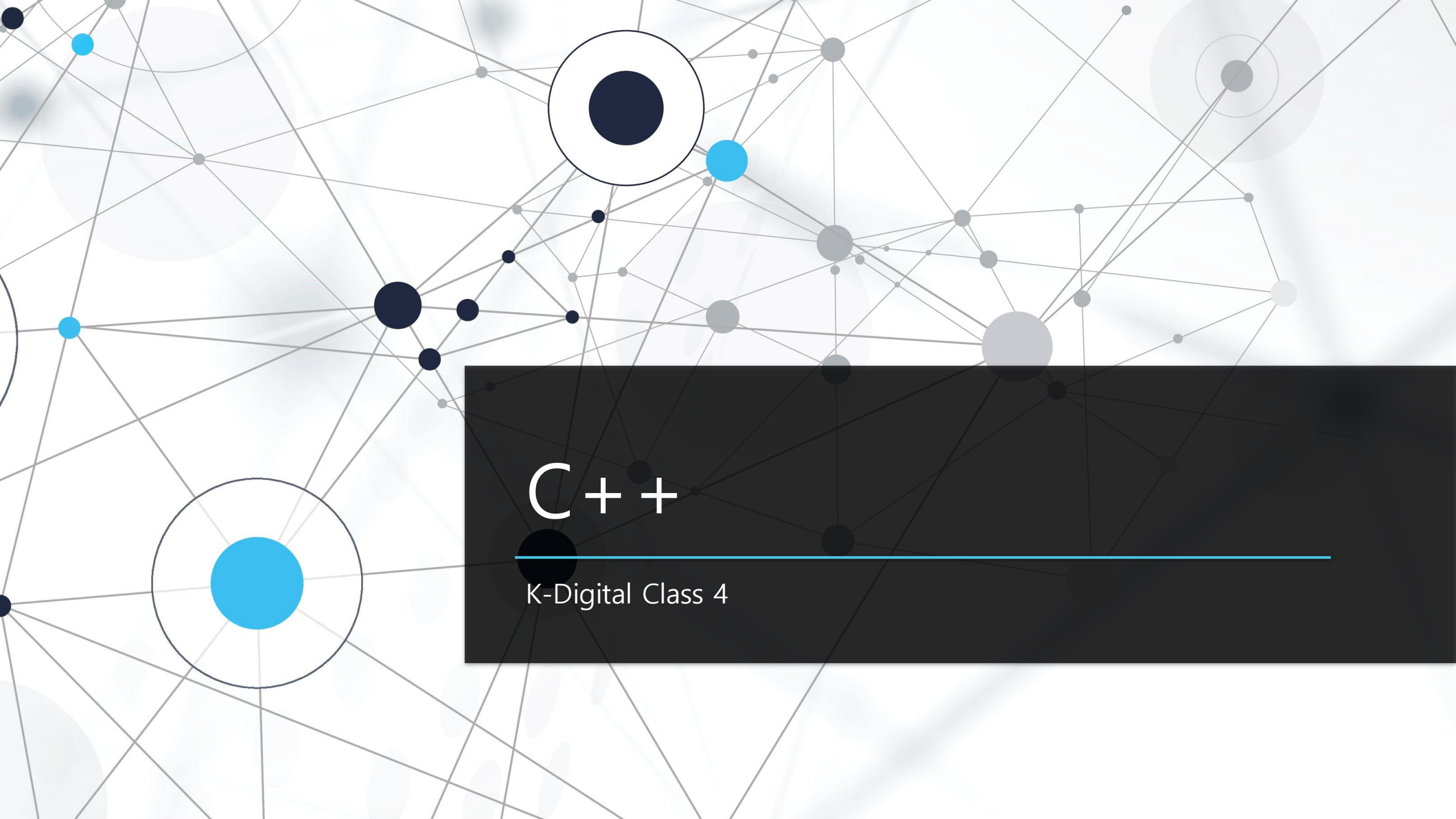


C++

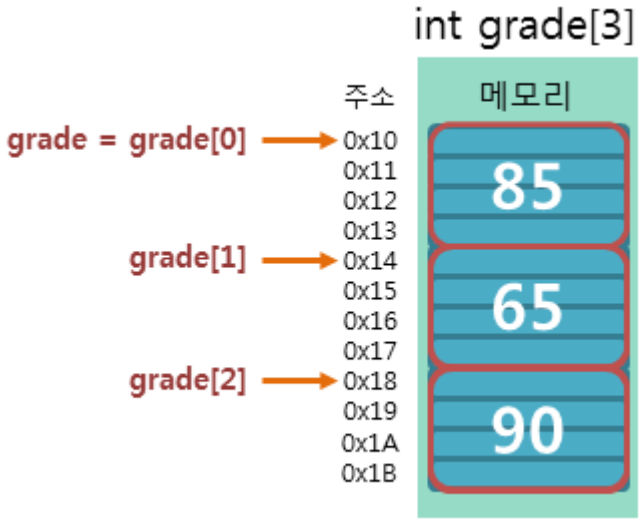
K-Digital Class 4

C++ ARRAYS AND POINTER

C++ Arrays

- 배열은 각 값에 대해 별도의 변수를 선언하는 대신 단일 변수에 여러 값을 저장하는 데 사용된다.
- 배열을 선언하려면 변수 유형을 정의하고 배열 이름과 대괄호를 지정하고 저장해야 하는 요소 수를 지정한다.
  - type arrayName[length];

```
string cars[4];
```



C++ ARRAYS AND POINTER

C++ Omit Arrays

- 배열의 크기를 지정 하지 않고도 선언할 수 있다.
- 그러나 삽입된 요소만큼만 크기가 커진다.

```
string cars[] = {"Volvo", "BMW", "Ford"}; // size of array is always 3
```

C++ ARRAYS AND POINTER

C++ References

- Reference variable(참조 변수)는 기존 변수에 대한 “참조”이며 & 연산자로 생성된다.

```
string food = "Pizza";  // food variable
string &meal = food;    // reference to food
```

C++ ARRAYS AND POINTER

C++ Memory Address

- 이전 페이지의 예에서는 & 연산자를 사용하여 참조 변수를 생성했다.
- 그러나 변수의 메모리 주소를 얻는 데에도 사용할 수 있다. 메모리에 변수가 저장되는 위치이다.
- C++에서 변수를 생성하면 변수에 메모리 주소가 할당된다. 그리고 변수에 값을 할당하면 이 메모리 주소에 저장된다.
- access하려면 & 연산자를 사용하 변수가 저장된 위치를 나타낸다.

```
string food = "Pizza";

cout << &food; // Outputs 0x6dfed4
```

메모리 주소를 아는 것이 왜 유용한가?
참조와 포인터는 C++에서 중요하다.
왜냐하면 컴퓨터 메모리의 데이터를 조작할 수 있는 능력을 제공하기 때문이다.
이러한 것은 code를 줄이고 성능을 향상시킬 수 있다.
이 두 가지 기능은 C++를 Python 및 Java와 같은 다른 프로그래밍 언어와 차별화하는 요소 중 하나이다.

C++ ARRAYS AND POINTER

C++ Pointer

- 변수가 선언되면 그 값을 저장하는 데 필요한 메모리의 특정 위치(메모리 주소)가 할당된다.
- 일반적으로 C++ 프로그램은 변수가 저장되는 정확한 메모리 주소를 능동적으로 결정하지 않는다.
- 다행히도 그 작업은 프로그램이 실행되는 환경에 맡겨집니다. 일반적으로 런타임 시 특정 메모리 위치를 결정하는 것은 운영 체제이다.
- 그러나 프로그램이 런타임 중에 변수에 대한 특정 위치에 있는 데이터 셀에 access(접근)하기 위해 변수의 주소를 얻어오는 것이 유용할 수 있다.
- C++에서 포인터(pointer)란 메모리의 주소 값을 저장하는 변수이며, 포인터 변수라고도 부른다.
- Char형 변수가 문자를 저장하고, int형 변수가 정수를 저장하는 것처럼 포인터는 주소 값을 저장하는 데 사용된다.

C++ ARRAYS AND POINTER

Address-of (주소) operator - &

- 변수의 주소는 변수 이름 앞에 주소 연산자라고 하는 ampersand 기호 (&)를 붙여서 얻을 수 있다.

Dereference(역 참조) operator-*

- 다른 변수의 주소를 저장하는 변수를 포인터라고 한다.
- 포인터는 주소가 저장되어 있는 변수를 "가리키는" 것이다.

```
string food = "Pizza";  // A string variable
string* ptr = &food;  // A pointer variable that stores the address of food
```

C++ ARRAYS AND POINTER

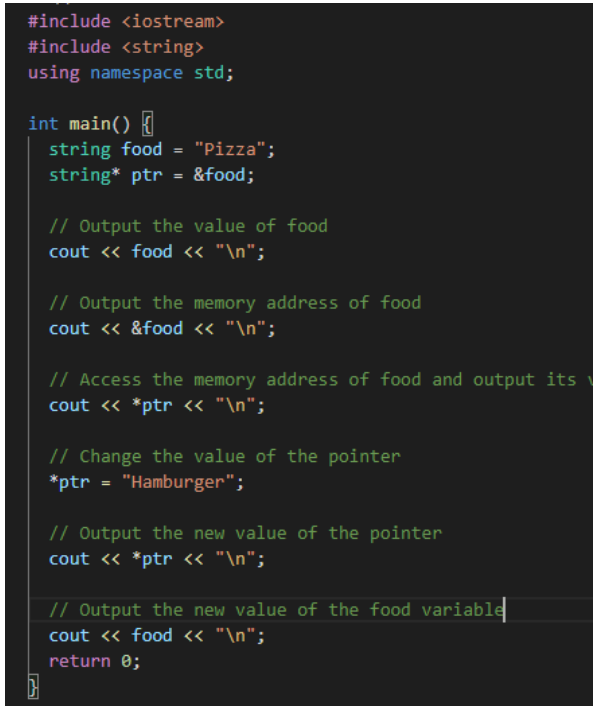
Declaring pointers

- 포인터 변수 선언의 일반적인 형식은 다음과 같다.
- type * name;
  - int * number;
  - char * character;
  - double * decimals;

C++ ARRAYS AND POINTER

C++ Modify Pointers

- 포인터의 값을 변경할 수도 있다.
- 그러나 이렇게 하면 원래 변수의 값도 변경된다.



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

int main() {
  string food = "Pizza";
  string* ptr = &food;

  // Output the value of food
  cout << food << "\n";

  // Output the memory address of food
  cout << &food << "\n";

  // Access the memory address of food and output its
  cout << *ptr << "\n";

  // Change the value of the pointer
  *ptr = "Hamburger";

  // Output the new value of the pointer
  cout << *ptr << "\n";

  // Output the new value of the food variable
  cout << food << "\n";
  return 0;
}
```

C++ ARRAYS AND POINTER

C++ Pointers and arrays

- 배열의 개념은 포인터의 개념과 관련이 있다. 실제로 배열의 첫 번째 요소는 포인터와 매우 유사하게 작동하며 실제로 배열은 항상 적절한 유형의 포인터로 묵시적으로 변환될 수 있다.
- 포인터와 배열은 동일한 작업 집합을 지원하며 둘 다 동일한 의미를 갖는다. 주요 차이점은 포인터에는 새 주소를 할당할 수 있지만 배열에는 할당할 수 없다는 것이다.

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

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

C++ ARRAYS AND POINTER

C++ Pointers and arrays

- 배열에 대한 장에서 대괄호([])는 배열 요소의 인덱스를 지정하는 것이다. 사실 이 괄호는 오프셋 연산자로 알려진 역참조 연산자이다. 그들은 뒤따르는 변수를 *처럼 역참조하지만 역참조되는 주소에 대괄호 사이의 숫자도 추가한다.
- 이 두 표현식은 포인터인 경우 뿐 아니라 가 배열인 경우에도 동일하고 유효하다. 배열인 경우 해당 이름을 첫 번째 요소에 대하여 포인터처럼 사용할 수 있음을 기억해야 한다.

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed to by (a+5) = 0
```

C++ ARRAYS AND POINTER

C++ Pointer initialization

- 포인터를 선언한 후 참조 연산자(*)를 사용하기 전에 포인터는 반드시 초기화되어야 한다.
- 초기화하지 않은 채로 참조 연산자를 사용하게 되면, 어딘지 알 수 없는 메모리 장소에 값을 저장하는 것이 된다.

```cpp
#include <iostream>

using namespace std;

int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

C++ ARRAYS AND POINTER

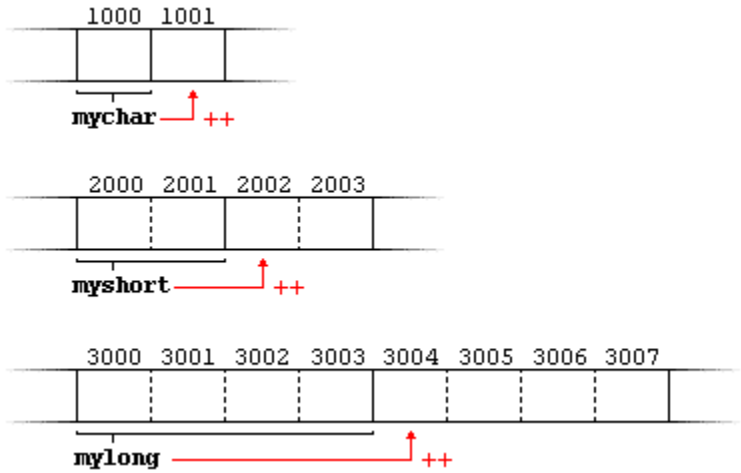
C++ Pointer arithmetics(pointer 연산)

- 포인터에서 산술 연산을 수행하는 것은 일반 정수 유형에서 수행하는 것과 약간 다르다.
- 우선 덧셈과 뺄셈 연산만 허용되며 나머지 연산은 포인터의 세계에서 의미가 없다.
- 그러나 덧셈과 뺄셈은 포인터가 가리키는 데이터 유형의 크기에 따라 포인터에 대해 다른 동작을 한다.
- 기본 데이터 유형은 크기가 서로 다른 것을 배웠다. 예를 들어: char의 크기는 항상 1바이트이고 short는 일반적으로 그보다 크며 int와 long은 훨씬 더 큰다. 이들의 정확한 크기는 시스템에 따라 다르다. 예를 들어, 사용하는 시스템에서 char은 1바이트, short는 2바이트, long은 4를 사용한다고 가정해 보겠다.

C++ ARRAYS AND POINTER

C++ Pointer arithmetics(pointer 연산)

- char *mychar;
- short *myshort;
- long *mylong;

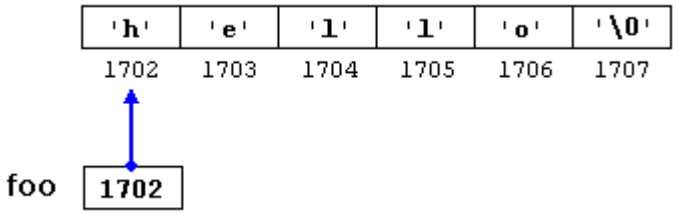


- ++mychar;
- ++myshort;
- ++mylong;

C++ ARRAYS AND POINTER

C++ Pointers and string literals

- 문자열 literal은 null로 끝나는 문자 sequence를 포함하는 배열이다.
- 문자열 literal은 cout에 직접 사용되어 문자열을 초기화하고 문자 배열을 초기화하는 데 사용되었다.
  - const char * foo = "hello";
- "hello"에 대한 literal 표현으로 배열을 선언하고 첫 번째 요소에 대한 포인터가 foo에 할당된다. " hello " 가 주소 1702에서 시작하는 메모리 위치에 저장되어 있다고 상상하면 이전 선언을 다음과 같이 나타낼 수 있다.



C++ ARRAYS AND POINTER

C++ Pointers and string literals

- 여기에서 foo는 포인터이고 주소 값 1702를 포함하고 ' h ' 나 " hello " 가 아니라 실제로 1702가 이 두 가지의 주소이다.
- 포인터 foo는 일련의 문자를 가리킨다. 그리고 포인터와 배열은 표현식에서 본질적으로 같은 방식으로 동작하기 때문에 foo는 null로 끝나는 문자 sequence의 배열과 같은 방식으로 문자에 액세스하는 데 사용할 수 있다.

- *(foo+4)
- foo[4]
- 두 표현식 모두 'o'(배열의 다섯 번째 요소) 값을 갖는다.

C++ ARRAYS AND POINTER

C++ Pointers to pointers(이중 포인터)

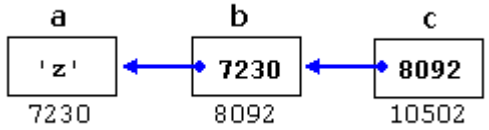
- C++에서는 포인터를 가리키는 포인터를 사용할 수 있다.
- 이 포인터는 차례로 데이터(또는 다른 포인터도 가리킴)를 가리킨다.
- 구문에는 포인터 선언의 각 간접 참조 수준에 대해 별표(*)가 필요하다.

- char a;
- char * b;
- char ** c;
- a = 'z';
- b = &a;
- c = &b;

C++ ARRAYS AND POINTER

C++ Pointers to pointers(이중 포인터)

- 선택된 메모리 위치를 7230, 8092 및 10502의 각 변수에 대해 임의로 가정하면 다음과 같이 나타낼 수 있다.
- 이 예에서 새로운 점은 포인터에 대한 포인터인 변수 c이며 세 가지의 다른 간접 참조 수준에서 사용할 수 있으며 각각은 다른 주소 값이다.



C++ ARRAYS AND POINTER

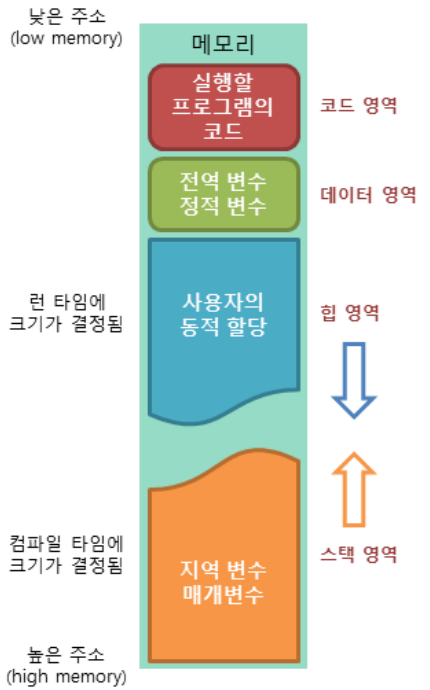
C++ void pointer

- void 유형의 포인터는 특별한 유형의 포인터입니다. C++에서 void는 type(유형)이 없음을 나타낸다. 따라서 void 포인터는 유형이 없는 값을 가리키는 포인터이다.
- 이것은 정수 값이나 부동 소수점에서 문자열에 이르기까지 모든 데이터 유형을 가리킬 수 있으므로 void 포인터에 큰 유연성을 제공한다.
- 그 대가로 그들은 큰 한계를 가지고 있다.
  - 그들이 가리키는 데이터는 직접 역 참조될 수 없으며(역 참조할 유형이 없기 때문에 논리적임), 이러한 이유로 void 포인터의 모든 주소는 구체적인 포인터 유형을 가리키는 다른 포인터 유형으로 변환되어야 한다.

C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

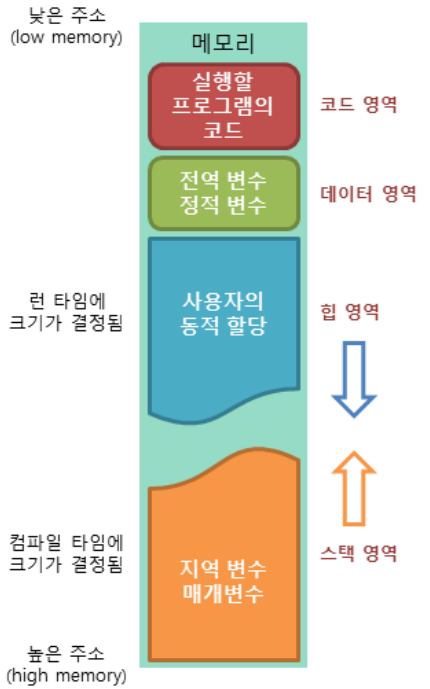
- 프로그램이 실행되기 위해서는 먼저 프로그램이 메모리에 로드(load)되어야 한다.
- 또한, 프로그램에서 사용되는 변수들을 저장할 메모리도 필요하다.
- 따라서 컴퓨터의 운영체제(Operating System)는 프로그램의 실행을 위해 다양한 메모리 공간을 제공하고 있다.
- 프로그램이 운영체제로부터 할당 받는 대표적인 메모리 공간은 다음과 같다.
  - 코드(code) 영역
  - 데이터(data) 영역
  - 스택(stack) 영역
  - 힙(heap) 영역



C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

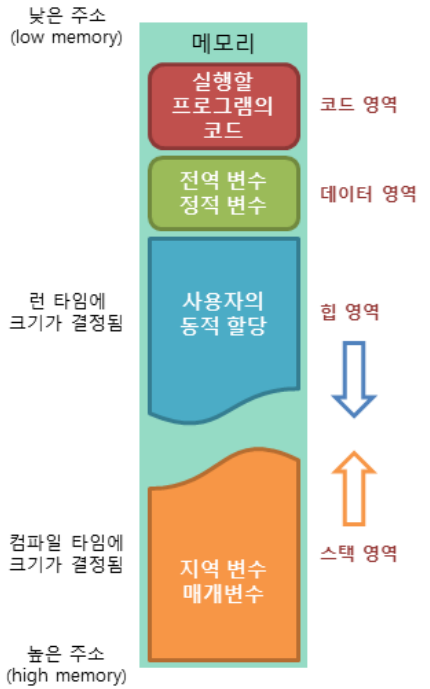
- Code 영역
  - 메모리의 코드(code) 영역은 실행할 프로그램의 코드가 저장되는 영역으로 텍스트(code) 영역이라고도 부른다.
  - CPU는 코드 영역에 저장된 명령어를 하나씩 가져가서 처리하게 된다.
- Data 영역
  - 메모리의 데이터(data) 영역은 프로그램의 전역 변수와 정적(static) 변수가 저장되는 영역이다.
  - 데이터 영역은 프로그램의 시작과 함께 할당되며, 프로그램이 종료되면 소멸한다.



C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

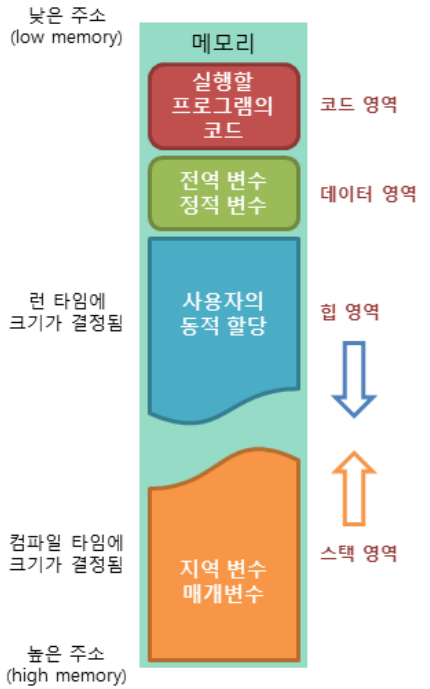
- Stack 영역
  - 메모리의 스택(stack) 영역은 함수의 호출과 관계되는 지역 변수와 매개변수가 저장되는 영역이다.
  - 스택 영역은 함수의 호출과 함께 할당되며, 함수의 호출이 완료되면 소멸한다.
  - 이렇게 스택 영역에 저장되는 함수의 호출 정보를 스택 프레임(stack frame)이라고 한다.
  - 스택 영역은 푸시(push) 동작으로 데이터를 저장하고, 팝(pop) 동작으로 데이터를 인출한다.
  - 이러한 스택은 후입선출(LIFO, Last-In First-Out) 방식에 따라 동작하므로, 가장 늦게 저장된 데이터가 가장 먼저 인출된다.
  - 스택 영역은 메모리의 높은 주소에서 낮은 주소의 방향으로 할당된다.



C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

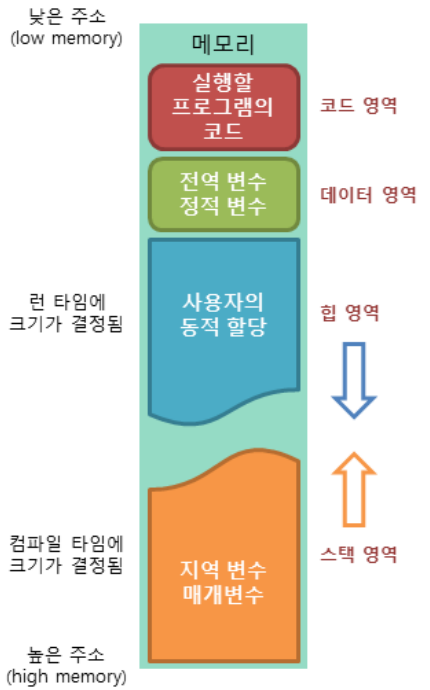
- Stack frame
  - 함수가 호출되면 스택에는 함수의 매개변수, 호출이 끝난 뒤 돌아갈 반환 주소 값, 함수에서 선언된 지역 변수 등이 저장된다.
  - 이렇게 스택 영역에 차례대로 저장되는 함수의 호출 정보를 스택 프레임(stack frame)이라고 한다.
  - 이러한 스택 프레임 덕분에 함수의 호출이 모두 끝난 뒤에, 해당 함수가 호출되기 이전 상태로 되돌아갈 수 있다.



C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

- Heap 영역
  - 메모리의 힙(heap) 영역은 사용자가 직접 관리할 수 있는 '그리고 해야만 하는' 메모리 영역이다.
  - 힙 영역은 사용자에 의해 메모리 공간이 동적으로 할당되고 해제된다.
  - 힙 영역은 메모리의 낮은 주소에서 높은 주소의 방향으로 할당된다.



C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- 데이터 영역과 스택 영역에 할당되는 메모리의 크기는 컴파일 타임(compile time)에 미리 결정된다.
- 하지만 Heap 영역의 크기는 프로그램이 실행되는 도중인 런 타임(run time)에 사용자가 직접 결정하게 된다.
- 이렇게 런 타임에 메모리를 할당 받는 것을 메모리의 동적 할당(dynamic allocation)이라고 한다.
- 포인터의 가장 큰 목적은 런 타임에 이름 없는 메모리를 할당 받아 포인터에 할당하여, 할당 받은 메모리에 접근하는 것이다.
- C언어에서는 malloc() 함수 등의 라이브러리 함수를 제공하여 이러한 작업을 수행할 수 있게 해준다.
- C++에서도 C언어의 라이브러리 함수를 사용하여 메모리의 동적 할당 및 해제를 할 수 있다.
- 하지만 C++에서는 메모리의 동적 할당 및 해제를 위한 더욱 효과적인 방법을 new 연산자와 delete 연산자를 통해 제공한다.

C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- C malloc() 함수
  - Malloc() 함수는 프로그램이 실행 중일 때 사용자가 직접 힙 영역에 메모리를 할당할 수 있게 해줍니다.
  - Malloc() 함수의 원형은 다음과 같습니다.
  - <stdlib.h> 에 포함 된 C standard library이다.
  - malloc() 함수는 인수로 할당 받고자 하는 메모리의 크기를 바이트 단위로 전달받는다.
  - 이 함수는 전달받은 메모리 크기에 맞고, 아직 할당되지 않은 적당한 블록을 찾는다.
  - 이렇게 찾은 블록의 첫 번째 바이트를 가리키는 주소 값을 반환한다.
  - 힙 영역에 할당할 수 있는 적당한 블록이 없을 때에는 널 포인터를 반환한다.
  - 주소 값을 반환 받기 때문에 힙 영역에 할당된 메모리 공간으로 접근하려면 포인터를 사용해야 한다.

```
#include <stdlib.h>
void *malloc(size_t size);
```

C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- C free() 함수
  - free() 함수는 힙 영역에 할당 받은 메모리 공간을 다시 운영 체제로 반환해 주는 함수이다.
  - 데이터 영역이나 스택 영역에 할당되는 메모리의 크기는 컴파일 타임에 결정되어, 프로그램이 실행되는 내내 고정된다.
  - 하지만 메모리의 동적 할당으로 힙 영역에 생성되는 메모리의 크기는 런 타임 내내 변화된다.
  - 따라서 free() 함수를 사용하여 다 사용한 메모리를 해제해 주지 않으면, 메모리가 부족해지는 현상이 발생할 수 있다.
  - 이처럼 사용이 끝난 메모리를 해제하지 않아서 메모리가 부족해지는 현상을 메모리 누수(memory leak)라고 한다.
  - free() 함수의 원형은 다음과 같습니다.
  - free() 함수는 인수로 해제하고자 하는 메모리 공간을 가리키는 포인터를 전달받는다.
  - 인수의 타입이 void형 포인터로 선언되어 있으므로, 어떠한 타입의 포인터라도 인수로 전달될 수 있다.

```
#include <stdlib.h>
void free(void *ptr);
```

C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- calloc() 함수
  - calloc() 함수는 malloc() 함수와 마찬가지로 힙 영역에 메모리를 동적으로 할당해주는 함수이다.
  - 이 함수가 malloc() 함수와 다른 점은 할당하고자 하는 메모리의 크기를 두 개의 인수로 나누어 전달받는 점이다.
  - 또한, calloc() 함수는 메모리를 할당 받은 후에 해당 메모리의 모든 비트 값을 전부 0으로 초기화해 준다.
  - Calloc() 함수도 malloc() 함수와 마찬가지로 free() 함수를 통해 할당 받은 메모리를 해제해 주어야 한다.

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- realloc() 함수
  - realloc() 함수는 이미 할당된 메모리의 크기를 바꾸어 재할당할 때 사용하는 함수이다.
  - realloc() 함수의 첫 번째 인수는 크기를 바꾸고자 하는 메모리 공간을 가리키는 포인터를 전달받는다.
  - 두 번째 인수로는 해당 메모리 공간에 재할당할 크기를 전달한다.
  - 따라서 첫 번째 인수로 NULL이 전달되면, malloc() 함수와 정확히 같은 동작을 하게 된다.

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

C++ ARRAYS AND POINTER

C++ new and delete operator

- new operator
  - C++ 모든 데이터 유형에 대해 동적으로 메모리를 할당하기 위해 new 연산자를 사용하는 다음 일반 구문이 있다.
  - Type은 데이터에 맞는 포인터를 선언하기 위해, 두 번째 타입은 메모리의 종류를 지정하기 위해 사용된다.
  - 만약 사용할 수 있는 메모리가 부족하여 새로운 메모리를 만들지 못했다면, new 연산자는 널 포인터를 반환한다.
  - 또한, new 연산자를 통해 할당 받은 메모리는 따로 이름이 없으므로 해당 포인터로만 접근할 수 있게 된다.

- type* ptrName = new type;

C++ ARRAYS AND POINTER

C++ new and delete operator

- delete operator
  - C언어에서는 free() 함수를 이용하여 동적으로 할당 받은 메모리를 다시 운영체제로 반환한다.
  - 이와 마찬가지로 C++에서는 delete 연산자를 사용하여, 더는 사용하지 않는 메모리를 다시 메모리 공간에 돌려줄 수 있다.
  - C++에서 delete 연산자는 다음과 같은 문법으로 사용합니다.

- delete ptrName;

C++ ARRAYS AND POINTER

C++ new[] and delete[] operator

- new[] operator
  - 문자 배열, 즉 20자의 문자열에 대한 메모리를 할당한다고 가정하면, 예제와 같이 메모리를 동적으로 할당할 수 있다.

```
char* pvalue  = NULL;          // Pointer initialized with null
pvalue  = new char[20];        // Request memory for the variable
```

```
double** pvalue  = NULL;       // Pointer initialized with null
pvalue  = new double [3][4];   // Allocate memory for a 3x4 array
```

C++ ARRAYS AND POINTER

C++ new[] and delete[] operator

- delete[] operator
- 앞장에서 생성 한 배열을 제거하려면 다음과 같이 한다.

```
delete [] pvalue;            // Delete array pointed to by pvalue
```

```
delete [] pvalue;            // Delete array pointed to by pvalue
```

C++ ARRAYS AND POINTER

C++ Two-dimensional Arrays

- 2차원 배열이란 배열의 요소로 1차원 배열을 가지는 배열이다.
- C++에서는 2차원 배열을 나타내는 타입을 따로 제공하지 않는다.
- 대신에 1차원 배열의 배열 요소로 또 다른 1차원 배열을 사용하여 2차원 배열을 나타낼 수 있다.
- type(타입)은 배열 요소로 저장되는 변수의 타입을 설정한다.
- arrName은 배열이 선언된 후에 배열에 접근하기 위해 사용된다.

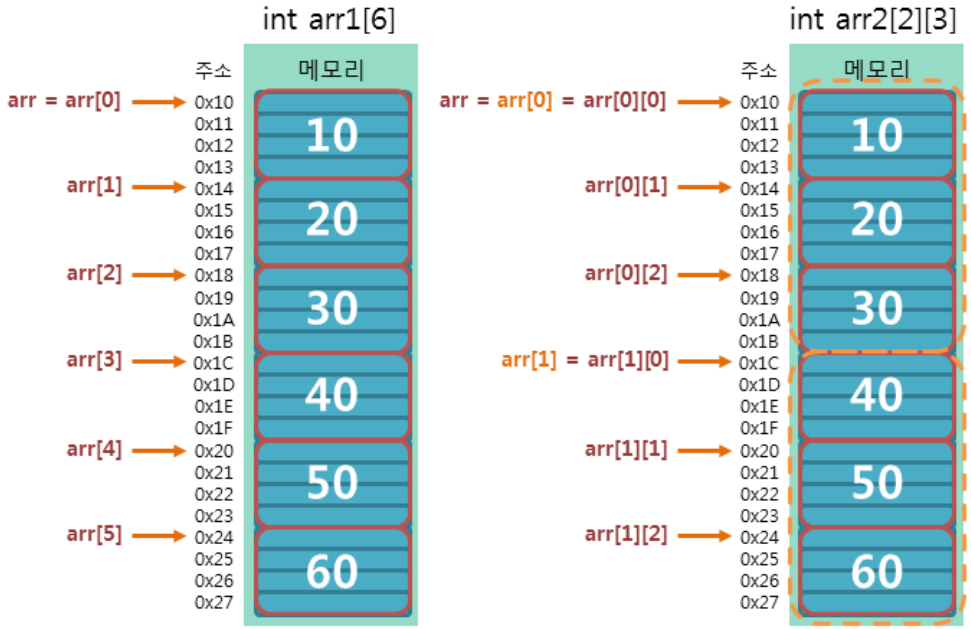
```
type arrayName [ x ][ y ];
```

C++ ARRAYS AND POINTER

C++ Two-dimensional Arrays

- 하지만 컴퓨터의 Memory는 입체적 공간이 아닌 선형 공간이므로 실제로는 다음 그림과 같이 저장된다.

```
int arr1[6] = {10, 20, 30, 40, 50, 60};
int arr2[2][3] = {10, 20, 30, 40, 50, 60};
```



C++ ARRAYS AND POINTER

Initializing Two-Dimensional Arrays

- 다차원 배열은 각 행에 대괄호로 묶인 값을 지정하여 초기화할 수 있다. 다음은 3행4열이 있는 배열의 선언과 함께 초기화 하는 방법이다.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

C++ ARRAYS AND POINTER

Initializing Two-Dimensional Arrays

- 행을 나타내는 중첩 중괄호는 선택 사항이다.
- 다음 초기화는 이전 예제와 동일하다.

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

C++ ARRAYS AND POINTER

C++ Pointers to pointers & Arrays

- [row][col] 크기의 2차원 배열
  - 원소를 col개 가지고 있는 1차원 배열이 row개 있는 것.
  - 1차원 배열의 주소가 row개 있는 것이나 마찬가지다.
  - 2차원 배열은 포인터의 배열이나 마찬가지다.

```
int main() {
int row = 3, col = 5;
int * row_1 = new int[col] {1, 2, 3, 4, 5}; // int 원소 5개의 1차원 동적 배열 row_1
int * row_2 = new int[col] {6, 7, 8, 9, 10}; // int 원소 5개의 1차원 동적 배열 row_2
int * row_3 = new int[col] {11, 12, 13, 14, 15}; // int 원소 5개의 1차원 동적 배열 row_2

int ** towd_array = new int* [row] {row_1, row_2, row_3};
// int* 포인터 원소 3개(= 1차원 배열이름 3개)의 2차원 동적 배열 2d_array
// 포인터 3개가 들어있는 배열(포인터)의 주소를 리턴받음
// 포인터의 포인터이므로 2d_array의 타입은 int **가 된다.

delete [] row_1;
delete [] row_2;
delete [] row_3;
delete [] towd_array;
```

C++ FUNCTION

- 함수는 호출될 때만 실행되는 코드 블록(code block)이다.
- 매개변수라고 하는 데이터를 함수에 전달할 수 있다.
- 함수는 특정 작업을 수행하는 데 사용되며 코드를 재사용하는 데 중요하다. 코드를 한 번 정의하고 여러 번 사용한다.
- 코드를 별도의 기능으로 나눌 수 있다. 코드를 다른 기능으로 나누는 방법은 사용자에게 달려 있지만 논리적으로 구분은 일반적으로 각 기능이 특정 작업을 수행하도록 한다.

C++ FUNCTION

Defining a Function

- C++는 코드를 실행하는 데 사용되는 main()과 같은 일부 미리 정의된 함수를 제공한다. 그러나 특정 작업을 수행하기 위해 고유한 기능을 만들 수도 있다.
- 함수를 생성(종종 선언이라고도 함)하려면 함수 이름을 지정하고 그 뒤에 괄호()를 붙인다.

```
void myFunction() {
  // code to be executed
}
```

C++ FUNCTION

Call a Function

- C++ 함수를 생성하는 동안 함수가 수행해야 하는 작업에 대한 정의를 제공한다. 함수를 사용하려면 해당 함수를 호출해야 한다. 프로그램이 함수를 호출하면 프로그램 제어가 호출된 함수로 이전된다.
- 호출된 함수는 정의된 작업을 수행하고 return 문이 실행되거나 함수 끝 닫는 중괄호에 도달하면 프로그램 제어를 다시 호출한 메인 프로그램 또는 호출한 함수로 반환한다.

Inside `main`, call `myFunction()`:

```
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

C++ FUNCTION

C++ Function Parameters

- 정보는 매개변수로 함수에 전달할 수 있습니다. 매개변수는 함수 내에서 변수 역할을 한다.
- 매개변수는 함수 이름 뒤에 괄호 안에 지정된다. 원하는 만큼 매개변수를 추가할 수 있다. 쉼표(,)로 구분하기만 하면 된다.

Syntax

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

C++ FUNCTION

C++ Function Default Parameter Value

- 등호(=)를 사용하여 기본 매개변수 값을 사용할 수도 있다.
- 인수 없이 함수를 호출하면 기본값 ("Norway")이 사용된다.

```
void myFunction(string country = "Norway") {
  cout << country << "\n";
}

int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}

// Sweden
// India
// Norway
// USA
```

C++ FUNCTION

C++ Function Multiple Parameters

- 함수 내에서 원하는 만큼 매개변수를 추가할 수 있다.

```
void myFunction(string fname, int age) {
  cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Liam Refsnes. 3 years old.
// Jenny Refsnes. 14 years old.
// Anja Refsnes. 30 years old.
```

C++ FUNCTION

C++ Function Return Values

- 이전 예에서 사용된 void 키워드는 함수가 값을 반환하지 않아야 함을 나타낸다.
- 함수가 값을 반환하도록 하려면 void 대신 데이터 type(유형)(예: int, string 등)을 사용하고 함수 내에서 return 키워드를 사용한다.

```
int myFunction(int x) {
  return 5 + x;
}

int main() {
  cout << myFunction(3);
  return 0;
}

// Outputs 8 (5 + 3)
```

C++ FUNCTION

C++ Function Pass By Reference Argument

- 이전 예제에서는 함수에 매개변수를 전달할 때 일반 변수를 사용했다.
- 함수에 대한 매개변수를 참조로 전달할 수도 있다. 이는 인수 값을 변경해야 할 때 유용할 수 있다.

```
void swapNums(int &x, int &y) {
  int z = x;
  x = y;
  y = z;
}

int main() {
  int firstNum = 10;
  int secondNum = 20;

  cout << "Before swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  // Call the function, which will change the values of firstNum and secondNum
  swapNums(firstNum, secondNum);

  cout << "After swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  return 0;
}
```

C++ FUNCTION

C++ Function Overloading

- 디폴트 인수가 인수의 개수를 달리하여 같은 함수를 호출하는 것이라면, 함수 오버로딩(overloading)은 같은 이름의 함수를 중복하여 정의하는 것을 의미한다.
- C++에서 새롭게 추가된 함수 오버로딩은 여러 함수를 하나의 이름으로 연결해 준다.
- 즉, 함수 오버로딩이란 같은 일을 처리하는 함수를 매개변수의 형식을 조금씩 달리하여, 하나의 이름으로 작성할 수 있게 해주는 것이다.
- 이와 같은 함수 오버로딩은 객체 지향 프로그래밍의 특징 중 바로 다형성(polymorphism)의 구현이다.

C++ FUNCTION

C++ Function signature

- 함수 오버로딩의 핵심은 바로 함수 시그니처(function signature)에 있다.
- 함수 시그니처란 함수의 원형에 명시되는 매개변수 리스트를 가리킨다.
- 만약 두 함수가 매개변수의 개수와 그 타입이 모두 같다면, 이 두 함수의 시그니처는 같다고 할 수 있다.
- 즉, 함수의 오버로딩은 서로 다른 시그니처를 갖는 여러 함수를 같은 이름으로 정의하는 것이라고 할 수 있다.

C++ FUNCTION

C++ Function signature

```
int myFunction(int x)
float myFunction(float x)
double myFunction(double x, double y)
```

```
int add(int x, int y)
{
    return x + y;
}

float add(float x, float y)
{
    return x + y;
}

int main()
{
    cout << "Adding Integer number : " << add(13, 67) << endl;
    cout << "Adding float number : " << add(13.43f, 67.56f) << endl;
    std::cout << "Hello World!\n";

    return 0;
}
```

C++ FUNCTION

C++ Function pointer

- 프로그램에서 정의된 함수는 프로그램이 실행될 때 모두 메인 메모리에 올라간다.
- 이때 함수의 이름은 메모리에 올라간 함수의 시작 주소를 가리키는 포인터 상수(constant pointer)가 된다.
- 이렇게 함수의 시작 주소를 가리키는 포인터 상수를 함수 포인터(function pointer)라고 부른다.
- 함수 포인터의 포인터 타입은 함수의 반환 값과 매개변수에 의해 결정된다.
- 즉 함수의 원형을 알아야만 해당 함수에 맞는 함수 포인터를 만들 수 있다.

포인터 상수(constant pointer)란 포인터 변수가 가리키고 있는 주소 값을 변경할 수 없는 포인터를 의미하며, 상수 포인터(pointer to constant)란 상수를 가리키는 포인터를 의미한다.

C++ FUNCTION

C++ Function pointer

- 함수의 원형(Function Prototype)
- void Func(int, int);

- 함수 포인터
- void (*ptr_func)(int, int);

C++ NAMESPACE

Namespaces in C++

- C++에서는 변수, 함수, 구조체, 클래스 등을 서로 구분하기 위해서 이름으로 사용되는 다양한 내부 식별자(identifier)를 가지고 있다.
- 하지만 프로그램이 복잡해지고 여러 라이브러리가 포함될수록 내부 식별자 간에 충돌할 가능성도 그만큼 커진다.
  - 외부 라이브러리 사용시 현재 모듈과 같은 함수가 존재 한다면 함수 이름의 충돌이 발생해 링크가 되지 않는다.
- 이러한 이름 충돌 문제를 C++에서는 네임스페이스(namespace)를 통해 해결하고 있다.
- C++에서 네임스페이스(namespace)란 내부 식별자에 사용될 수 있는 유효 범위를 제공하는 선언적 영역을 의미한다.

C++ NAMESPACE

Defining a Namespace

- C++에서는 namespace 키워드를 사용하여 사용자가 새로운 네임스페이스를 정의할 수 있다.
- 이러한 네임스페이스는 전역 위치 뿐만 아니라 다른 네임스페이스 내에서도 정의될 수 있다.
- 하지만 code 블록 내에서는 정의될 수 없으며, 기본적으로 외부 연결을 가지게 된다.
- 일반적으로 namespace는 헤더 파일에서 정의되며, 언제나 새로운 이름을 추가할 수 있도록 개방되어 있다.
- C++에서는 전역 네임스페이스(global namespace)라고 하는 파일 수준의 선언 영역이 존재한다.
- 일반적으로 식별자의 네임스페이스가 명시되지 않으면, 전역 네임스페이스에 자동으로 포함되게 된다.
- 또한, C++ 표준 라이브러리 타입과 함수들은 std 네임스페이스 또는 그 속에 중첩된 네임스페이스에 선언되어 있다.

```
namespace namespace_name {
   // code declarations
}
```

```
name::code;  // code could be variable or function.
```

C++ NAMESPACE

Accessing a Namespace

- namespace를 정의한 후에는 해당 네임스페이스로 접근할 수 있는 방법이 필요하다.
- namespace에 접근하기 위해서는 범위 지정 연산자(::, scope resolution operator)를 사용하여, 해당 이름을 특정 namespace로 제한하면 된다.

```
std::cout << "Hello World!\n";
```

C++ NAMESPACE

Access to simplified namespaces

- namespace에 속한 이름을 사용할 때마다 매번 범위 지정 연산자(scope resolution operator;::)를 사용하여 이름을 제한하는 것은 매우 불편하다.
- 또한, 길어진 코드로 인해 가독성 또한 떨어지게 된다.
- C++에서는 이러한 불편함을 해소할 수 있도록 다음과 같은 방법을 제공하고 있다.
  - using 지시자(directive)
  - using 선언(declaration)

C++ NAMESPACE

using 지시자(directive)

- using 지시자는 명시한 namespace에 속한 이름을 모두 가져와 범위 지정 연산자를 사용하지 않고도 사용할 수 있게 해준다.
- 전역 범위에서 사용된 using 지시자는 해당 namespace의 모든 이름을 전역적으로 사용할 수 있게 만들어 준다.
- 또한, 블록 내에서 사용된 using 지시자는 해당 블록에서만 해당 namespace의 모든 이름을 사용할 수 있게 해준다.

```
using namespace 네임스페이스이름;
```

C++ NAMESPACE

using 선언(declaration)

- using 지시자가 명시한 namespace의 모든 이름을 사용할 수 있게 했다면, using 선언은 단 하나의 이름만을 범위 지정 연산자를 사용하지 않고도 사용할 수 있게 해준다.
- 또한, using 지시자와 마찬가지로 using 선언이 나타나는 선언 영역에서만 해당 이름을 사용할 수 있게 해준다.

```
using 네임스페이스이름::이름;
```

C++ NAMESPACE

Discontiguous Namespaces(불연속 네임스페이스)

- namespace는 여러 부분으로 정의될 수 있으므로 네임스페이스는 별도로 정의된 부분의 합으로 구성된다. namespace의 개별 부분은 여러 파일에 분산될 수 있다.
- 따라서 namespace의 한 부분에 다른 파일에 정의된 이름이 필요한 경우 해당 이름을 계속 선언해야 한다.
- 다음 네임스페이스 정의를 작성하면 새 네임스페이스를 정의하거나 기존 네임스페이스에 새 요소를 추가한다.

```
namespace namespace_name {
   // code declarations
}
```

C++ NAMESPACE

Nested Namespaces(중첩된 네임스페이스)

- 네임스페이스는 다음과 같이 다른 네임스페이스 안에 하나의 네임스페이스를 정의할 수 있는 중첩될 수 있다.

```
namespace namespace_name1 {
   // code declarations
   namespace namespace_name2 {
      // code declarations
   }
}
```

C++ NAMESPACE

Nested Namespaces(중첩된 네임스페이스)

- 다음과 같이 확인 연산자를 사용하여 중첩된 네임스페이스의 멤버에 액세스할 수 있다.

```
// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;

// to access members of namespace:name1
using namespace namespace_name1;
```

C++ DATA STRUCTURE

- C/C++ 배열을 사용하면 같은 종류의 여러 데이터 항목을 결합하는 변수를 정의할 수 있지만 구조체(structure)는 다른 종류의 데이터 항목을 결합할 수 있는 또 다른 사용자 정의 데이터 유형이다.
- 배열이 같은 타입의 변수 집합이라고 한다면, 구조체는 다양한 타입의 변수 집합을 하나의 타입으로 나타낸 것이다.
- 이때 구조체를 구성하는 변수를 구조체의 멤버(member) 또는 멤버 변수(member variable)라고 한다.
- C/C++의 구조체는 변수 뿐만 아니라 함수까지도 멤버로 가질 수 있다.
- 또한, C/C++의 구조체는 타입일 뿐만 아니라, 객체 지향 프로그래밍의 핵심이 되는 클래스(class)의 기초가 되었다.

C++ DATA STRUCTURE

Defining a Structure

- 구조체를 정의하려면 struct 문을 사용한다.
- Struct 문은 프로그램에 대해 둘 이상의 멤버가 있는 새 데이터 유형을 정의한다.

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

C++ DATA STRUCTURE

Accessing Structure Members

- 구조체의 모든 멤버에 액세스하려면 멤버 액세스 연산자(.)를 사용한다.
- 멤버 액세스 연산자는 구조체 변수 이름과 Access하려는 구조체 멤버 사이의 마침표(.)로 코딩한다.
- 구조체 유형의 변수를 정의하려면 struct 키워드를 사용한다.

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

using namespace std;

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   cout << "Book 1 title : " << Book1.title <<endl;
   cout << "Book 1 author : " << Book1.author <<endl;
   cout << "Book 1 subject : " << Book1.subject <<endl;
   cout << "Book 1 id : " << Book1.book_id <<endl;

   // Print Book2 info
   cout << "Book 2 title : " << Book2.title <<endl;
   cout << "Book 2 author : " << Book2.author <<endl;
   cout << "Book 2 subject : " << Book2.subject <<endl;
   cout << "Book 2 id : " << Book2.book_id <<endl;

   return 0;
}
```

C++ DATA STRUCTURE

Structures as Function Arguments

- 다른 변수나 포인터를 전달할 때와 매우 유사한 방식으로 구조체를 함수 인수로 전달할 수 있다.

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books book );

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   printBook( Book1 );

   // Print Book2 info
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book ) {
   cout << "Book title : " << book.title <<endl;
   cout << "Book author : " << book.author <<endl;
   cout << "Book subject : " << book.subject <<endl;
   cout << "Book id : " << book.book_id <<endl;
}
```

C++ DATA STRUCTURE

Pointers to Structures

- 변수에 대한 포인터를 정의하는 것과 매우 유사한 방식으로 구조에 대한 포인터를 정의할 수 있습니다.

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books *book );

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // Book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info, passing address of structure
   printBook( &Book1 );

   // Print Book1 info, passing address of structure
   printBook( &Book2 );

   return 0;
}

// This function accept pointer to structure as parameter.
void printBook( struct Books *book ) {
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

C++ DATA STRUCTURE

The typedef Keyword

- 구조체를 정의하는 더 쉬운 방법이 있거나 생성한 유형을 “alias”(별칭)으로 지정할 수 있다.

```
typedef struct {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} Books;

Books Book1, Book2;
```

C++ DATA STRUCTURE

Size of Structure

- 구조체의 크기는 멤버 변수들의 크기에 따라 결정된다.
- 하지만 구조체의 크기가 언제나 멤버 변수들의 크기 총합과 일치하는 것은 아니다.

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

struct TypeSize
{
    char a;
    int b;
    double c;
};

int main(void)
{
    cout << "구조체 TypeSize의 각 멤버의 크기는 다음과 같습니다." << endl;
    cout << sizeof(char) << ", " << sizeof(int) << ", " << sizeof(double) << endl << endl;

    cout << "구조체 TypeSize의 크기는 다음과 같습니다." << endl;
    cout << sizeof(TypeSize);
    return 0;
}
```

C++ DATA STRUCTURE

Size of Structure

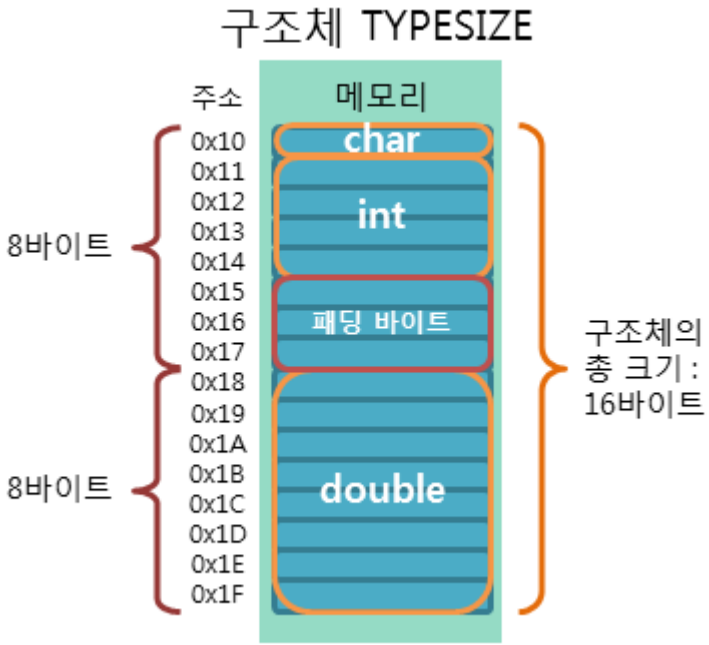
- 위의 예제에서 구조체 멤버 변수의 크기는 각각 1, 4, 8바이트이다.
- 하지만 구조체의 크기는 멤버 변수들의 크기 총합인 13바이트가 아니라 16바이트가 된다.
- 구조체를 메모리에 할당할 때 컴파일러는 프로그램의 속도 향상을 위해 바이트 패딩(byte padding)이라는 규칙을 이용한다.
- 구조체는 다양한 크기의 타입을 멤버 변수로 가질 수 있는 타입이다.
- 하지만 컴파일러는 메모리의 접근을 쉽게 하려고 크기가 가장 큰 멤버 변수를 기준으로 모든 멤버 변수의 메모리 크기를 맞추게 된다.
- 이것을 바이트 패딩이라고 하며, 이때 추가되는 바이트를 패딩 바이트(padding byte)라고 한다.

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

struct TypeSize
{
    char a;
    int b;
    double c;
};

int main(void)
{
    cout << "구조체 TypeSize의 각 멤버의 크기는 다음과 같습니다." << endl;
    cout << sizeof(char) << ", " << sizeof(int) << ", " << sizeof(double) << endl << endl;

    cout << "구조체 TypeSize의 크기는 다음과 같습니다." << endl;
    cout << sizeof(TypeSize);
    return 0;
}
```

C++ DATA STRUCTURE

Size of Structure

- 예제에서는 크기가 가장 큰 double형 타입의 크기인 8바이트가 기준이 된다.
- 맨 처음 char형 멤버 변수를 위해 8바이트가 할당되며, 할당되는 1바이트를 제외한 7바이트가 남게 된다.
- 그 다음 int형 멤버 변수는 남은 7바이트보다 작으므로, 그대로 7바이트 중 4바이트를 할당하고 3바이트가 남게 된다.
- 마지막 double형 멤버 변수는 8바이트인데 남은 공간은 3바이트 뿐이므로 다시 8바이트를 할당 받는다.
- 따라서 이 구조체의 크기는 총 16바이트가 되며, 그 중에서 패딩 바이트는 3바이트가 된다.



C++ DATA STRUCTURE

Size of Structure

- 구조체 안에 변수를 선언 할 때 크기가 작은 자료형 -> 크기가 큰 자료형 순으로 선언한다. 위의 예시에서 살펴 봤듯이, 구조체는 변수를 선언하는 순서가 중요하다. 따라서 크기가 작은 자료형부터 크기가 큰 자료형 순으로 구조체를 선언 했다면 패딩으로 인한 문제가 줄어든다.
- 메모리 절약을 생각한다면, 맨 처음 #pragma pack(push, 1)와 #pragma pack(pop)를 사용한다. #pragma pack(push, 1)를 사용하게 되면 구조체의 기본 단위가 1byte가 된다.(push 옆에 1이라고 지정했으므로). 따라서 패딩 문제가 발생하지 않는다.

```
#pragma pack(push, 1)
struct TypeSize
{
    char a;
    int b;
    double c;
};
#pragma pack(pop)
```

C++ OBJECT ORIENTED

C++ What is OOP(Object Oriented Programming)?

- OOP는 Object Oriented Programming(객체 지향 프로그래밍)의 약자이다.
- 절차적 프로그래밍(Procedural Programming)은 Data에 대한 작업을 수행하는 절차 혹은 함수를 작성하는 것이고 객체 지향 프로그래밍은 데이터와 함수를 모두 포함하는 객체(Object)를 만드는 것이다.

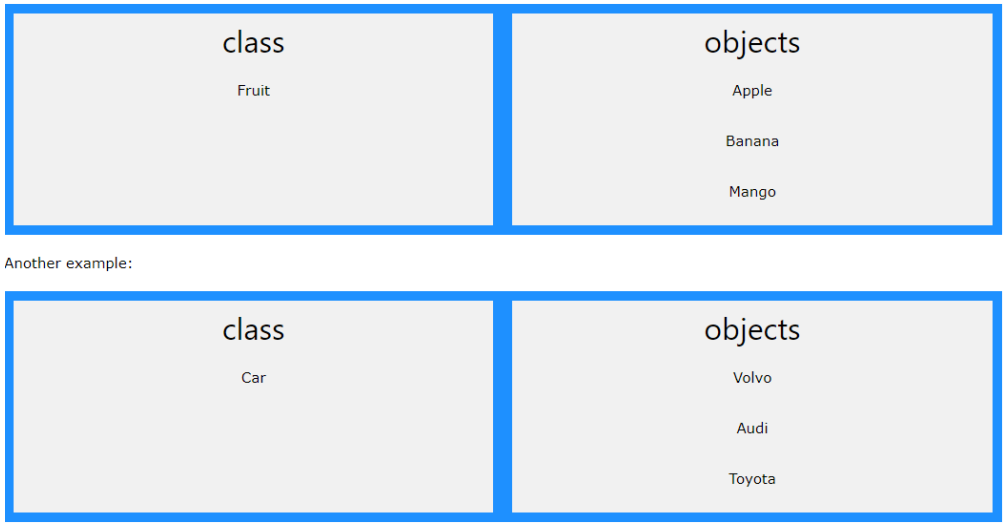
- 객체 지향 프로그래밍은 절차적 프로그래밍에 비해 몇 가지 장점이 있다.
  - OOP는 더 빠르고 쉽게 실행할 수 있다.
  - OOP는 프로그램에 대한 명확한 구조를 제공한다.
  - OOP는 C++ 코드 DRY ("Don't Repeat Yourself")를 유지하는 데 도움이 되며 코드를 유지 관리, 수정 및 디버그하기 쉽게 만든다.
  - OOP를 사용하면 더 적은 코드와 더 짧은 개발 시간으로 완전히 재사용 가능한 application을 만들 수 있다.

DRY(Don't Repeat Yourself) 원칙은 코드 반복을 줄이는 것이다. 애플리케이션에 공통적인 코드를 추출하여 한 곳에 배치하고 반복하지 않고 재사용해야 한다.

C++ OBJECT ORIENTED

C++ What are Classes and Objects?

- class와 object는 객체 지향 프로그래밍의 두 가지 주요 측면이다.
- 클래스와 객체의 차이점을 보려면 다음 그림을 참고 한다.
- class는 object의 템플릿(template)이고 object는 class의 instance이다. 개별 객체가 생성되면 클래스에서 모든 변수와 기능을 상속한다.



C++ OBJECT ORIENTED

C++ Classes and Objects

- C++는 Object Oriented Programming(객체 지향 프로그래밍) 언어(Language)이다.
- C++의 모든 것은 속성(attribute) 및 메서드(method)와 함께 class 및 object와 연관이 있다.
- 예를 들어 실생활에서 자동차는 물건(Object)이다. 자동차는 무게와 색상과 같은 속성(attribute)과 drive, brake와 같은 방법(method)을 가지고 있습니다.
- Attribute와 method는 기본적으로 class에 속하는 변수(variable)와 함수(function)이다. 이들은 “class member " 라고 한다.
- class는 우리 프로그램에서 사용할 수 있는 사용자 정의 데이터 유형이며 object constructor(객체 생성자)를 통하여 작동한다.

C++ OBJECT ORIENTED

C++ Class Definitions

- 클래스를 정의할 때 데이터 유형에 대한 청사진을 정의한다. 이것은 실제로 데이터를 정의하지 않지만 클래스 이름이 의미하는 바, 즉 클래스의 객체가 구성되는 것과 그러한 객체에서 수행할 수 있는 작업을 정의한다.
- 클래스 정의는 class 키워드로 시작하고 그 뒤에 클래스 이름이 온다. 한 쌍의 중괄호({})로 묶인 클래스 본문. 클래스 정의 뒤에는 세미콜론이나 선언 목록이 와야 한다.

C++ OBJECT ORIENTED

C++ Class Definitions

```
class MyClass {        // The class
  public:              // Access specifier
    int myNum;         // Attribute (int variable)
    string myString;   // Attribute (string variable)
};
```

- class 키워드는 MyClass라는 클래스를 만드는 데 사용된다.
- public 키워드는 클래스의 멤버(속성 및 메서드)가 클래스 외부에서 액세스할 수 있도록 지정하는 액세스 지정자 (access specifier)이다.
- 클래스 내부에는 int 변수 myNum과 string 변수 myString이 있다. 변수가 클래스 내에서 선언되면 속성(attribute)이라고 한다.마지막으로 클래스 정의를 세미콜론 ;으로 끝낸다.

C++ OBJECT ORIENTED

Define C++ Objects

- 클래스는 객체에 대한 청사진(blueprint)을 제공하므로 기본적으로 객체는 클래스에서 생성된다.
- 기본 유형의 변수를 선언하는 것과 정확히 같은 종류의 선언으로 클래스의 객체를 선언한다.

Create an object called " myObj " and access the attributes:

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

C++ OBJECT ORIENTED

Define C++ Objects

- 클래스는 객체에 대한 청사진(blueprint)을 제공하므로 기본적으로 객체는 클래스에서 생성된다.
- 기본 유형의 변수를 선언하는 것과 정확히 같은 종류의 선언으로 클래스의 객체를 선언한다.

Create an object called " myObj " and access the attributes:

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

C++ OBJECT ORIENTED

Multiple Objects

- 한 클래스의 여러 객체를 만들 수 있다.

```
// Create a Car class with some attributes
class Car {
  public:
    string brand;
    string model;
    int year;
};

int main() {
  // Create an object of Car
  Car carObj1;
  carObj1.brand = "BMW";
  carObj1.model = "X5";
  carObj1.year = 1999;

  // Create another object of Car
  Car carObj2;
  carObj2.brand = "Ford";
  carObj2.model = "Mustang";
  carObj2.year = 1969;

  // Print attribute values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

C++ OBJECT ORIENTED

Class Methods

- Method는 class에 속하는 function이다.
- 클래스에 속하는 함수를 정의하는 두 가지 방법이 있다.
  - Inside class definition.
  - Outside class definition.
- 속성에 액세스하는 것처럼 메서드에 액세스한다. 클래스의 객체를 만들고 점 구문(.)을 사용한다.

C++ OBJECT ORIENTED

Class Methods - Inside class definition

Inside Example

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod() {  // Method/function defined inside the class
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

C++ OBJECT ORIENTED

Class Methods - Outside class definition

- 클래스 정의 외부에서 함수를 정의하려면 클래스 내부에서 선언한 다음 클래스 외부에서 정의해야 합니다.
- 이것은 클래스 이름, scope resolution operator ::함수 이름을 차례로 지정하여 수행된다.

Outside Example

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod();   // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

C++ OBJECT ORIENTED

Class Methods – Parameter

- You can also add parameters.

Example

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

class Car {
  public:
    int speed(int maxSpeed);
};

int Car::speed(int maxSpeed) {
  return maxSpeed;
}

int main() {
  Car myObj; // Create an object of Car
  cout << myObj.speed(200); // Call the method with an argument
  return 0;
}
```

C++ OBJECT ORIENTED

C++ class Constructors(Default Constructor)

- C++의 객체 생성자는 클래스의 객체가 생성될 때 멤버 변수를 초기화 하며 자동으로 호출되는 특수 method이다.
- 생성자를 생성하려면 클래스와 동일한 이름을 사용하고 그 뒤에 괄호()를 사용한다. Default Constructor라고 한다.
- 생성자는 클래스와 이름이 같으며 항상 public이며 return 값이 없다.

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object of MyClass (this will call the constructor)
  return 0;
}
```

C++ OBJECT ORIENTED

C++ class Constructors Parameters

- 생성자는 속성의 초기 값을 설정하는 데 유용할 수 있는 매개변수(일반 함수와 마찬가지로)를 사용할 수 있다.

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

class Car {         // The class
  public:           // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z) {  // Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

C++ OBJECT ORIENTED

C++ class Constructors Parameters

- 함수와 마찬가지로 생성자도 클래스 외부에서 정의할 수 있다.
- 먼저 클래스 내부에서 생성자를 선언한 다음 클래스 이름, scope resolution :: operator를 사용하여, 생성자 이름(클래스와 동일)을 지정하여 클래스 외부에서 정의한다.

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

class Car {         // The class
  public:           // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

C++ OBJECT ORIENTED

C++ The Class Destructor(파괴자 or 소멸자)

- 소멸자(destructor)는 클래스의 객체가 범위를 벗어날 때마다 또는 삭제 식(delete expression) 즉 삭제 시 해당 클래스의 객체에 대한 포인터에 대하여 적용될 때마다 실행되는 클래스의 특수 멤버 함수이다.
- 한마디로 객체가 메모리에 반납될 때 실행되는 함수이다.
- 소멸자는 접두사(~)가 붙은 클래스와 정확히 같은 이름을 가지며 값을 반환하거나 매개변수를 사용할 수 없다. 소멸자는 파일 닫기, 메모리 해제 등과 같이 프로그램에서 나오기 전에 리소스를 해제하는 데 매우 유용할 수 있다.

```
#include <iostream>

using namespace std;
class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
Line::~Line(void) {
   cout << "Object is being deleted" << endl;
}
void Line::setLength( double len ) {
   length = len;
}
double Line::getLength( void ) {
   return length;
}

// Main function for the program
int main() {
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

C++ OBJECT ORIENTED

C++ The Class Destructor(파괴자 or 소멸자)

- 소멸자의 호출.
  - C++에서 소멸자의 호출 시기는 컴파일러가 알아서 처리하게 된다.
  - C++에서 객체가 선언된 메모리 영역별로 소멸자가 호출되는 시기는 다음과 같다.

메모리 영역	소멸자 호출 시기
데이터(data) 영역	해당 프로그램이 종료될 때
스택(stack) 영역	해당 객체가 정의된 블록을 벗어날 때
힙(heap) 영역	delete를 사용하여 해당 객체의 메모리를 반환할 때
임시 객체	임시 객체의 사용을 마쳤을 때

C++ OBJECT ORIENTED

C++ Copy Constructor(복사생성자)

- 얕은 복사와 깊은 복사
  - 깊은 복사(deep copy) - 새롭게 생성하는 변수에 다른 변수의 값을 대입하기 위해서는 대입 연산자(=)를 사용하면 된다.
    - int x = 10;
  - 얕은 복사(shallow copy)
    - 새롭게 생성하는 객체에 또 다른 객체의 값을 대입하기 위해서도 대입 연산자(=)를 사용할 수 있다.
      - Book web_book("HTML과 CSS", 350);
      - Book html_book = web_book;
    - 하지만 대입 연산자를 이용한 객체의 대입은 얕은 복사(shallow copy)로 수행됩니다.
    - 얕은 복사(shallow copy)란 값을 복사하는 것이 아닌, 값을 가리키는 포인터를 복사하는 것이다.
    - 따라서 변수의 생성에서 대입 연산자를 이용한 값의 복사는 문제가 되지 않지만, 객체에서는 문제가 발생할 수도 있다.
    - 특히 객체의 멤버가 메모리 공간의 힙(heap) 영역을 참조할 경우에는 문제가 발생한다.

C++ OBJECT ORIENTED

C++ Copy Constructor(복사생성자)

- C++에서 복사 생성자란 자신과 같은 클래스 타입의 다른 객체에 대한 참조(reference)를 인수로 전달받아, 그 참조를 가지고 자신을 초기화하는 방법이다.
- 복사 생성자는 새롭게 생성되는 객체가 원본 객체와 같으면서도, 완전한 독립성을 가지게 해준다.
- 왜냐하면, 복사 생성자를 이용한 대입은 깊은 복사(deep copy)를 통한 값의 복사이기 때문이다.
- 복사 생성자는 다음과 같은 상황에서 주로 사용된다.
  - 1. 객체가 함수에 인수로 전달될 때
  - 2. 함수가 객체를 반환 값으로 반환할 때
  - 3. 새로운 객체를 같은 클래스 타입의 기존 객체와 똑같이 초기화할 때

```
Book(const Book&);
```

C++ OBJECT ORIENTED

C++ Copy Constructor(복사생성자) – this

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

class Book
{
private:
    int current_page_;
    void set_percent();
public:
    Book(const string& title, int total_page);
    string title_;
    int total_page_;
    double percent_;
    void Move(int page);
    void Open();
    void Read();
    const Book& ThickerBook(const Book&);   // ThickerBook() 함수의 원형
};

int main(void)
{
    Book web_book("HTML과 CSS", 350);
    Book html_book("HTML 레퍼런스", 200);

    cout << web_book.ThickerBook(html_book).title_; // 더 두꺼운 책의 이름을 출력함.
    return 0;
}

Book::Book(const string& title, int total_page)
{
    title_ = title;
    total_page_ = total_page;
    current_page_ = 0;
    set_percent();
}

void Book::set_percent()
{
    percent_ = (double) current_page_ / total_page_ * 100;
}

const Book& Book::ThickerBook(const Book& comp_book)
{
    if (comp_book.total_page_ > this->total_page_)
    {
        return comp_book;
    }
    else
    {
        return *this;
    }
}
```

C++ OBJECT ORIENTED

C++ Overloading

- C++에서는 함수 이름 또는 동일한 범위(scope)의 연산자에 대해 둘 이상의 정의를 지정할 수 있다. 이를 각각 함수 오버로딩(Function overloading) 및 연산자 오버로딩(Operator Overloading)이라고 한다.
- 오버로드(Overload) 된 선언은 동일한 범위(scope)에서 이전에 선언된 선언과 동일한 이름으로 선언된 선언이다. 단, 두 선언 모두 인수(매개변수)가 다르고 정의(구현)가 분명히 다르다.
- 오버로드 된 함수 또는 연산자를 호출할 때 컴파일러는 함수 또는 연산자를 호출하는 데 사용한 인수 유형을 정의에 지정된 매개변수 유형과 비교하여 사용할 가장 적절한 정의를 결정한다. 가장 적절한 오버로드 된 함수 또는 연산자를 선택하는 프로세스를 오버로드 해결(overload resolution)이라고 한다.

C++ OBJECT ORIENTED

C++ Class Function Overloading

- 동일한 범위(Same Scope)에서 동일한 함수 이름에 대해 여러 정의를 가질 수 있다.
- 함수의 정의는 인수 목록의 인수 유형 또는 이수의 수가 따라 서로 달라야 한다.
- 반환 유형(return type)만 다른 함수 선언은 오버로드 할 수 없다.

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

class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

C++ OBJECT ORIENTED

Operators Overloading in C++

- C++에서 사용할 수 있는 대부분의 기본 제공 연산자를 재정의하거나 오버로드 할 수 있다. 따라서 프로그래머는 사용자 정의 유형(class etc)과 함께 연산자도 사용할 수 있다.
- 오버로드 된 연산자는 특수 이름을 가진 함수이다. Keyword " operator " 다음에 정의되는 연산자의 기호가 온다. 다른 함수와 마찬가지로 오버로드 된 연산자에는 반환 형식과 매개 변수 목록이 있다.
- Operator 오버로딩 할 연산자(매개변수목록)

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

class Position
{
private:
   double x_;
   double y_;
public:
   Position(double x, double y); // 생성자
   void Display();
   Position operator-(const Position& other);   // - 연산자 함수
};

int main(void)
{
   Position pos1 = Position(3.3, 12.5);
   Position pos2 = Position(-14.4, 7.8);
   Position pos3 = pos1 - pos2;

   pos3.Display();
   return 0;
}

Position::Position(double x, double y)
{
   x_ = x;
   y_ = y;
}

Position Position::operator-(const Position& other)
{
   return Position((x_ + other.x_)/2, (y_ + other.y_)/2);
}

void Position::Display()
{
   cout << "두 지점의 중간 지점의 좌표는 x좌표가 " << this->x_ << "이고, y좌표가 " << this->y_ << "입니다." << endl;
}
```

C++ OBJECT ORIENTED

Define Operators function in C++

- C++에서 연산자 함수를 정의하는 방법은 다음과 같이 두 가지 방법이 있다.
  - 클래스의 멤버 함수로 정의하는 방법
  - 전역 함수로 정의하는 방법
- 이 두 방법의 차이는 인수의 개수 뿐만 아니라 private 멤버에 대한 접근 여부도 있다.
- 연산자 함수를 전역 함수로 정의해야 할 경우, private 멤버에 대한 접근을 위해 C++에서 제공하는 friend 함수를 사용할 수 있다.

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

class Position
{
private:
    double x_;
    double y_;
public:
    Position(double x, double y); // 생성자
    void Display();
    friend Position operator-(const Position& pos1, const Position& pos2);  // - 연산자 함수
};

int main(void)
{
    Position pos1 = Position(3.3, 12.5);
    Position pos2 = Position(-14.4, 7.8);
    Position pos3 = pos1 - pos2;

    pos3.Display();
    return 0;
}

Position::Position(double x, double y)
{
    x_ = x;
    y_ = y;
}

Position operator-(const Position& pos1, const Position& pos2)
{
    return Position((pos1.x_ + pos2.x_)/2, (pos1.y_ + pos2.y_)/2);
}

void Position::Display()
{
    cout << "두 지점의 중간 지점의 좌표는 x좌표가 " << this->x_ << "이고, y좌표가 " << this->y_ << "입니다." << endl;
}
```

C++ OBJECT ORIENTED

오버로딩의 제약 사항

- C++에서 연산자를 오버로딩 할 때에는 다음과 같은 사항을 지켜야 한다.
  - 전혀 새로운 연산자를 정의할 수는 없다.
    - ex) 몫을 나타내기 위한 %%라는 연산자를 새롭게 정의할 수 없다.
  - 기본 타입을 다루는 연산자의 의미는 재정의할 수 없으며, 따라서 오버로딩 된 연산자의 피연산자 중 하나는 반드시 사용자 정의 타입이어야 한다.
    - Ex) 두 개의 double 형에 대한 덧셈 연산자(+)가 뺄셈을 수행하도록 오버로딩 할 수 없다.
  - 오버로딩 된 연산자는 기본 타입을 다루는 경우에 적용되는 피연산자의 수, 우선순위 및 그룹화를 준수해야 한다.
    - Ex) 나눗셈 연산자(/)는 이항 연산자이므로 단 항 연산자로 오버로딩 할 수 없다.
  - 오버로딩 된 연산자는 디폴트 인수를 사용할 수 없다.

C++ OBJECT ORIENTED

오버로딩의 제약 사항

- 오버로딩 할 수 없는 연산자

연산자	설명
::	범위 지정 연산자
.	멤버 연산자
.*	멤버 포인터 연산자
? :	삼항 조건 연산자
sizeof	크기 연산자
typeid	타입 인식
const_cast	상수 타입 변환
dynamic_cast	동적 타입 변환
reinterpret_cast	재해석 타입 변환
static_cast	정적 타입 변환

C++ OBJECT ORIENTED

오버로딩의 제약 사항

- 멤버 함수로만 오버로딩 할 수 있는 연산자
  - C++에서 다음 표의 연산자는 전역 함수가 아닌 멤버 함수로만 오버로딩 할 수 있다.

연산자	설명
=	대입 연산자
()	함수 호출
[]	배열 인덱스
->	멤버 접근 연산자

C++ OBJECT ORIENTED

Data hiding(정보 은닉)

- C++에서 구조체의 모든 멤버는 외부에서 언제나 접근할 수 있다.
- 하지만 클래스는 객체 지향 프로그래밍의 기본 규칙 중 하나인 정보 은닉에 대해서도 생각해야만 한다.
- 정보 은닉(data hiding)이란 사용자가 굳이 알 필요가 없는 정보는 사용자로부터 숨겨야 한다는 개념이다.
- 그렇게 함으로써 사용자는 언제나 최소한의 정보만으로 프로그램을 손쉽게 사용할 수 있게 된다.

C++ OBJECT ORIENTED

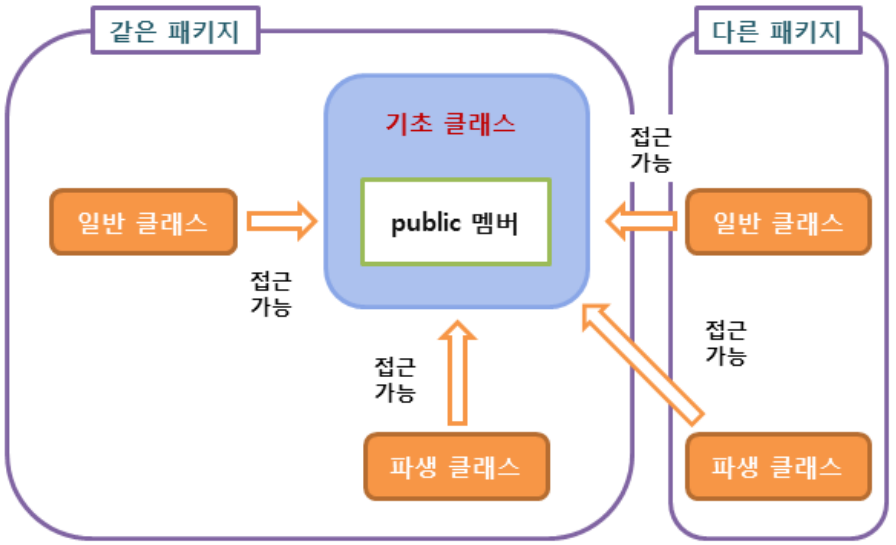
C++ class Access Specifiers

- C++에서는 이러한 정보 은닉을 위해 접근 제어(access control)라는 기능을 제공하고 있다.
- 접근 제어란 접근 제어 지시자(Access Control Specifiers)를 사용해 클래스 외부에서의 직접적인 접근을 허용하지 않는 멤버를 설정할 수 있도록 하여, 정보 은닉을 구체화하는 것을 의미한다.
- C++에서는 다음과 같은 세 가지의 접근 제어 지시자를 제공한다.
  - Public
  - Private
  - Protected
- 클래스의 기본 접근 제어 권한은 private이며, 구조체 및 공용체는 public이다.

C++ OBJECT ORIENTED

C++ class Access Specifiers – public

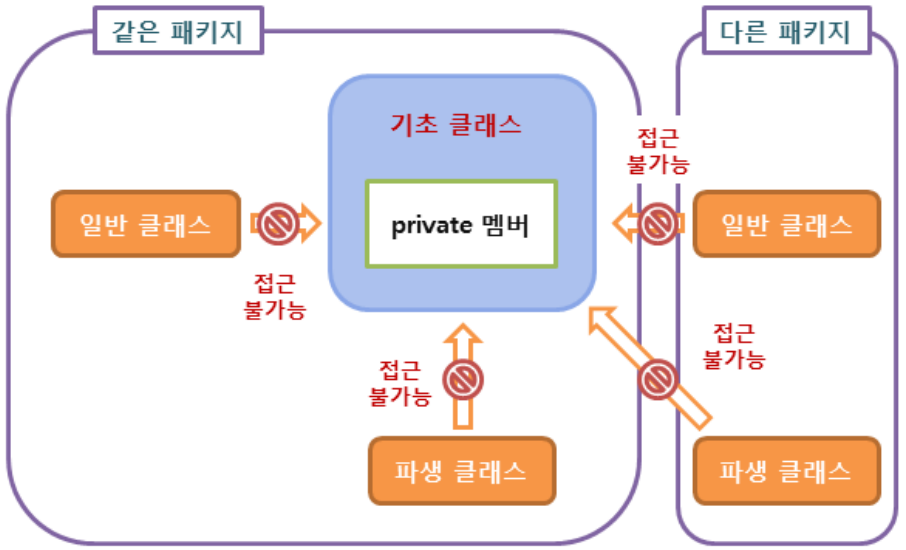
- public 접근 제어 지시자를 사용하여 선언된 클래스 멤버는 외부로 공개되며, 해당 객체를 사용하는 프로그램 어디에서나 직접 접근할 수 있다.
- 따라서 public 멤버 함수는 해당 객체의 private 멤버와 프로그램 사이의 인터페이스(interface) 역할을 하게 된다.
- 프로그램은 이러한 public 멤버 함수를 통해 해당 객체의 private 멤버에도 접근할 수 있도록 구현 해야한다.



C++ OBJECT ORIENTED

C++ class Access Specifiers – private

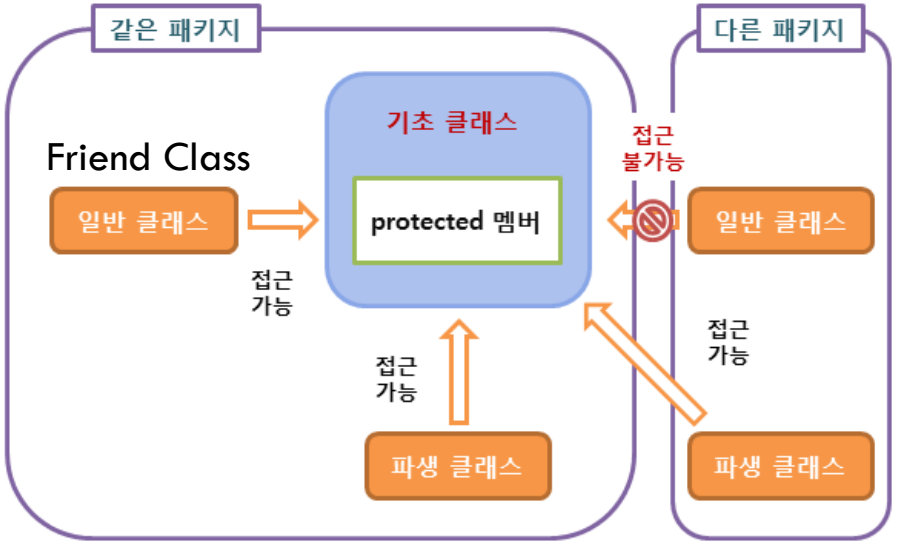
- private 접근 제어 지시자를 사용하면 선언된 클래스 멤버는 외부에 공개되지 않으며, 외부에서 직접 접근할 수도 없다.
- 프로그램은 private 멤버에 직접 접근할 수 없으며, 해당 객체의 public 멤버 함수를 통해서만 접근할 수 있다.
- 클래스의 기본 접근 제어 권한은 private로 설정되어 있으므로, 클래스 선언 시 private 접근 제어 지시자는 생략할 수 있다.
- 일반적으로 private 멤버는 public 인터페이스를 직접 구성하지 않는 클래스의 세부적인 동작을 구현하는 데 사용된다.



C++ OBJECT ORIENTED

C++ class Access Specifiers – protected

- C++ 클래스는 private 멤버로 정보를 은닉하고, public 멤버로 사용자나 프로그램과의 인터페이스를 구축한다.
- 여기에 파생 클래스(derived class)와 관련된 접근 제어 지시자가 하나 더 존재한다.
- Protected 멤버는 파생 클래스에 대해서는 public 멤버처럼 취급되며, 외부에서는 private 멤버처럼 취급된다.
- Protected 멤버에 접근할 수 있는 영역은 다음과 같다.
  - 이 멤버를 선언한 클래스의 멤버 함수
  - 이 멤버를 선언한 클래스의 friend class
  - 이 멤버를 선언한 클래스에서 public 또는 protected 접근 제어로 파생된 클래스(상속된 클래스)



C++ OBJECT ORIENTED

C++ class Access Specifiers

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

// Base class
class Employee  {
  protected:  // Protected access specifier
    int salary;
};

// Derived class
class Programmer: public Employee {
  public:
    int bonus;
    void setSalary(int s) {
      salary = s;
    }
    int getSalary() {
      return salary;
    }
};

int main() {
  Programmer myObj;
  myObj.setSalary(50000);
  myObj.bonus = 15000;
  cout << "Salary: " << myObj.getSalary() << "\n";
  cout << "Bonus: " << myObj.bonus << "\n";
  return 0;
}
```

C++ OBJECT ORIENTED

C++ Encapsulation

- 모든 C++ 프로그램은 다음 두 가지 기본 요소로 구성된다.
  - 프로그램 명령문(코드) – 이것은 작업을 수행하는 프로그램의 일부이며 이를 함수라고 한다.
  - 프로그램 데이터 – 데이터는 프로그램 기능의 영향을 받는 프로그램의 정보이다.
- 캡슐화는 데이터와 데이터를 조작하는 기능을 함께 묶고 외부 간섭과 오용으로부터 안전하게 유지하는 객체 지향 프로그래밍 개념이다.
- 데이터 캡슐화는 데이터 은닉의 중요한 OOP 개념으로 이어졌다.
- 데이터 캡슐화는 데이터를 묶는 mechanism이며, 이를 사용하는 함수와 데이터 추상화(data abstraction)는 인터페이스(interface)만 노출하고 구현 세부 사항을 사용자에게 숨기는 mechanism이다.

C++ OBJECT ORIENTED

C++ Encapsulation

- 캡슐화의 의미는 "민감한" 데이터가 사용자에게 숨겨져 있는지 확인하는 것이다. 이를 달성하려면 클래스 변수/속성을 private로 선언해야 한다(클래스 외부에서 액세스할 수 없음).
- 다른 사람들이 private 멤버의 값을 읽거나 수정하도록 하려면 public get 및 set 메서드를 제공 해야 한다.

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

class Employee {
  private:
    // Private attribute
    int salary;

  public:
    // Setter
    void setSalary(int s) {
      salary = s;
    }
    // Getter
    int getSalary() {
      return salary;
    }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

C++ OBJECT ORIENTED

C++ Why Encapsulation?

- 클래스 속성을 비공개로 선언하는 것은 좋은 습관이다(가능한 한 자주). 캡슐화는 다른 부분에 영향을 주지 않고 코드의 한 부분을 변경할 수 있으므로 데이터를 더 잘 제어할 수 있다.
- 데이터 보안이 향상된다.

C++ OBJECT ORIENTED

C++ Designing Strategy(디자인 전략)

- 우리 대부분은 클래스 멤버를 실제로 노출해야 하는 경우가 아니면 기본적으로 클래스 멤버를 비공개로 설정하는 방법을 배웠다.
- 프로그램 설계측면에서 Encapsulation은 아주 좋은 방법인 것이다.

C++ OBJECT ORIENTED

C++ Inheritance

- 객체 지향 프로그래밍에서 가장 중요한 개념 중 하나는 상속 개념이다.
- 상속을 통해 다른 클래스의 관점에서 클래스를 정의할 수 있으므로 Application을 더 쉽게 만들고 유지 관리할 수 있다.
- 이것은 또한 코드 기능을 재사용할 수 있는 기회를 제공하고 빠른 구현 시간의 기회를 제공한다.
- 클래스를 생성할 때 완전히 새로운 데이터 멤버와 멤버 함수를 작성하는 대신 프로그래머는 새 클래스가 기존 클래스의 멤버를 상속하도록 지정할 수 있다. 이 기존 클래스를 기본 클래스라고 하고 새 클래스를 파생 클래스라고 한다.
- 상속을 구현한다는 관계의 아이디어이다. 예를 들어, 포유류 IS-A 동물, 개 IS-A 포유류, 따라서 개 IS-A 동물 등이다.

C++ OBJECT ORIENTED

C++ Inheritance

- C++에서는 "상속 개념"을 두 가지 범주의 그룹이 있다.
  - Derived class(파생 클래스)(자식) - 다른 클래스에서 상속받은 클래스
  - Base class(기본 클래스)(부모) – 상속이 되는 클래스
- To inherit from a class, use the ':' symbol.
- access-specifier는 public, protected 또는 private 중 하나이고 base-class는 이전에 정의된 클래스(부모 클래스)의 이름이다. Access-specifier가 사용되지 않으면 기본적으로 private이다.

```
class derived-class: access-specifier base-class
```

C++ OBJECT ORIENTED

C++ Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

C++ OBJECT ORIENTED

Access Control and Inheritance

- 파생 클래스는 기본 클래스의 모든 비공개 멤버에 액세스할 수 있다.
- 따라서 derived class(파생 클래스)의 멤버 함수에 액세스할 수 없는 base class(기본 클래스) 멤버는 기본 클래스에서 private로 선언되어야 한다.
- 다음과 같은 방식으로 액세스할 수 있는 사용자에 따라 다양한 액세스 유형을 요약할 수 있다.
- 파생 클래스는 다음을 제외하고 모든 기본 클래스 메서드를 상속한다.
  - Constructors, destructors and copy constructors of the base class.
  - Overloaded operators of the base class.
  - The friend functions of the base class.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

C++ OBJECT ORIENTED

Type of Inheritance(상속 유형)

- 기본 클래스에서 클래스를 파생할 때 기본 클래스는 public, protected 또는 private 상속을 통해 상속될 수 있다. 상속 유형은 위에서 설명한 대로 액세스 지정자(Access Specifiers)에 의해 지정된다.
- 우리는 protected나 private 상속을 거의 사용하지 않지만, public 상속은 일반적으로 사용된다.

C++ OBJECT ORIENTED

Type of Inheritance(상속 유형)

- 다른 유형의 상속을 사용하는 동안 다음 규칙이 적용된다.
  - Public Inheritance - public 기본 클래스에서 클래스를 파생할 때 기본 클래스의 public 멤버는 파생 클래스의 public 멤버가 되고 기본 클래스의 protected 멤버는 파생 클래스의 protected 멤버가 된다. 기본 클래스의 private 멤버는 파생 클래스에서 직접 액세스할 수 없지만 기본 클래스의 public 및 protected 멤버에 대한 호출을 통해 액세스할 수 있다.
  - Protected Inheritance - protected 기본 클래스에서 파생할 때 기본 클래스의 public 및 protected 멤버는 파생 클래스의 protected 멤버가 된다.
  - Private Inheritance - private 기본 클래스에서 파생할 때 기본 클래스의 public 및 protected 멤버는 파생 클래스의 private 멤버가 된다.

C++ OBJECT ORIENTED

C++ Multilevel Inheritance(다단계 상속)

- 클래스는 이미 다른 클래스에서 파생된 한 클래스에서 파생될 수도 있다.

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

// Parent class
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};

// Child class
class MyChild: public MyClass {
};

// Grandchild class
class MyGrandChild: public MyChild {
};

int main() {
  MyGrandChild myObj;
  myObj.myFunction();
  return 0;
}
```

C++ OBJECT ORIENTED

C++ Multiple Inheritance(다중상속)

- 클래스는 쉼표(,)로 구분된 목록을 사용하여 둘 이상의 base class에서 파생될 수도 있다.
- 이것을 다중 상속이라 한다.

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

// Base class
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class.\n" ;
    }
};

// Another base class
class MyOtherClass {
  public:
    void myOtherFunction() {
      cout << "Some content in another class.\n" ;
    }
};

// Derived class
class MyChildClass: public MyClass, public MyOtherClass {
};

int main() {
  MyChildClass myObj;
  myObj.myFunction();
  myObj.myOtherFunction();
  return 0;
}
```

C++ OBJECT ORIENTED

C++ Class Friend

- C++에서 객체의 private 멤버에는 해당 객체의 public 멤버 함수를 통해서만 접근할 수 있다고 했다.
- 하지만 경우에 따라서는 해당 객체의 멤버 함수가 아닌 함수도 private 멤버에 접근해야만 할 경우가 발생한다.
- 이럴 때마다 매번 private 멤버에 접근하기 위한 새로운 public 멤버 함수를 작성하는 것은 매우 비효율적이다.
- 따라서 C++에서는 이러한 경우를 위해 friend라는 새로운 접근 제어 키워드를 제공한다.
- Friend는 지정한 대상에 한해 해당 객체의 모든 멤버에 접근할 수 있는 권한을 부여해 준다.
- 이러한 friend 키워드는 전역 함수, 클래스, 멤버 함수의 세 가지 형태로 사용할 수 있다.

C++ OBJECT ORIENTED

C++ Definition of Friend Function

- C++에서 friend 함수는 friend 키워드를 사용하여 다음과 같이 선언한다.
- 이렇게 선언된 friend 함수는 클래스 선언부에 그 원형이 포함되지만, 클래스의 멤버 함수는 아니다.
- 이러한 friend 함수는 해당 클래스의 멤버 함수는 아니지만, 멤버 함수와 같은 접근 권한을 가지게 된다.
- friend 키워드는 함수의 원형에서만 사용해야 하며, 함수의 정의에서는 사용하지 않는다.
- Friend 선언은 클래스 선언부의 public, private, protected 영역 등 어디에나 위치할 수 있으며, 위치에 따른 차이는 전혀 없다.
- 클래스 내부에서만 써야할 멤버들이 다른 곳에서 계속 접근을 허용하게 되면 데이터 보호도 어렵고 캡슐화를 지향하는 객체지향적 설계라고 보기 어렵다. friend는 가급적 남발하지 않고 필요한 경우에만 사용해 주는 것이 좋다.

- friend className funcName(매개변수);

C++ OBJECT ORIENTED

C++ Definition of Friend Function

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

class Rect
{
private:
    double height_;
    double width_;
public:
    Rect(double height, double width);  // 생성자
    void DisplaySize();
    Rect operator*(double mul) const;
    friend Rect operator*(double mul, const Rect& origin);  // 프렌드 함수
};

int main(void)
{
    Rect origin_rect(10, 20);
    Rect rect01 = origin_rect * 2;
    Rect rect02 = 3 * origin_rect;

    rect01.DisplaySize();
    rect02.DisplaySize();
    return 0;
}

Rect::Rect(double height, double width)
{
    height_ = height;
    width_ = width;
}

void Rect::DisplaySize()
{
    cout << "이 사각형의 높이는 " << this->height_ << "이고, 너비는 " << this->width_ << "입니다." << endl;
}

Rect Rect::operator*(double mul) const
{
    return Rect(height_ * mul, width_ * mul);
}

Rect operator*(double mul, const Rect& origin)
{
    return origin * mul;
}
```

C++ OBJECT ORIENTED

C++ Definition of Friend Class

- friend class className

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

class PeopleA{
private:
    string name;
    int height;
    friend class PeopleB; //friend 클래스 선언
public:
    //생성자
    PeopleA(string name, int height) {
        this->name = name;
        this->height = height;
    }
};

class PeopleB {
public:
    //friend 선언으로 인해 PeopleA의 private 멤버에 접근가능
    void info_A(PeopleA a) {
        cout << "이름 : " << a.name << endl;
        cout << "신장 : " << a.height << endl;
    }
};

void main() {
    PeopleA a("홍길동",170);
    PeopleB b;
    b.info_A(a);
}
```

C++ OBJECT ORIENTED

C++ Class static member variables

- C++에서 static member란 클래스에는 속하지만, 객체 별로 할당되지 않고 클래스의 모든 객체가 공유하는 멤버를 의미한다.
- 멤버 변수가 정적(static)으로 선언되면, 해당 클래스의 모든 객체에 대해 하나의 데이터만이 유지 관리된다.
- 정적 멤버 변수는 클래스 영역에서 선언되지만, 정의는 파일 영역에서 수행된다.
- 이러한 정적 멤버 변수는 외부 연결(external linkage)을 가지므로, 여러 파일에서 접근할 수 있다.
- 정적 멤버 변수에도 클래스 멤버의 접근 제한 규칙이 적용되므로, 클래스의 멤버 함수나 friend만이 접근할 수 있다.
- 하지만 정적 멤버 변수를 외부에서도 접근할 수 있게 하고 싶으면, 정적 멤버 변수를 public 영역에 선언하면 된다.

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

class Person
{
private:
    string name_;
    int age_;
public:
    static int person_count_;   // 정적 멤버 변수의 선언
    Person(const string& name, int age);    // 생성자
    ~Person() { person_count_--; }          // 소멸자
    void ShowPersonInfo();
};

int Person::person_count_ = 0;  // 정적 멤버 변수의 정의 및 초기화

int main(void)
{
    Person hong("길동", 29);
    hong.ShowPersonInfo();
    Person lee("순신", 35);
    lee.ShowPersonInfo();

    return 0;
}

Person::Person(const string& name, int age)
{
    name_ = name;
    age_ = age;
    cout << ++person_count_ << " 번째 사람이 생성되었습니다." << endl;
}

void Person::ShowPersonInfo()
{
    cout << "이 사람의 이름은 " << name_ << "이고, 나이는 " << age_ << "살입니다." << endl;
}
```

C++ OBJECT ORIENTED

C++ Class static member functions

- C++에서는 클래스의 멤버 함수도 정적(static)으로 선언할 수 있다.
- 이렇게 선언된 정적 멤버 함수는 해당 클래스의 객체를 생성하지 않고도, 클래스 이름만으로 호출할 수 있다.
- 정적 멤버 함수는 정적 멤버 변수를 선언하는 방법과 같이 static 키워드를 사용하여 선언한다.
  - 객체를 생성하지 않고 클래스 이름만으로 호출할 수 있다.
  - 객체를 생성하지 않으므로, this 포인터를 가지지 않는다.
  - 특정 객체와 결합하지 않으므로, 정적 멤버 변수밖에 사용할 수 없다

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

class Person
{
private:
    string name_;
    int age_;
public:
    static int person_count_;   // 정적 멤버 변수의 선언
    static int person_count();  // 정적 멤버 함수의 선언
    Person(const string& name, int age);    // 생성자
    ~Person() { person_count_--; }          // 소멸자
    void ShowPersonInfo();
};

int Person::person_count_ = 0;  // 정적 멤버 변수의 정의 및 초기화

int main(void)
{
    Person hong("길동", 29);
    Person lee("순신", 35);
    cout << "현재까지 생성된 총 인원 수는 " << Person::person_count() << "명입니다." << endl;

    return 0;
}

Person::Person(const string& name, int age)
{
    name_ = name;
    age_ = age;
    cout << ++person_count_ << " 번째 사람이 생성되었습니다." << endl;
}

void Person::ShowPersonInfo()
{
    cout << "이 사람의 이름은 " << name_ << "이고, 나이는 " << age_ << "살입니다." << endl;
}

int Person::person_count()      // 정적 멤버 함수의 정의
{
    return person_count_;
}
```

C++ OBJECT ORIENTED

C++ Polymorphism(다형성)

- 다형성이라는 단어는 많은 형태를 갖는다는 의미이다. 일반적으로 다형성은 클래스의 계층(hierarchy)이 있고 상속으로 관련되어 있을 때 발생한다.
- 상속을 통해 다른 클래스의 속성(attribute)과 메서드(method)를 상속할 수 있다. 다형성은 이러한 방법을 사용하여 다른 작업을 수행한다. 이를 통해 우리는 다양한 방식으로 단일 작업을 수행할 수 있다.

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

// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n" ;
    }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n" ;
    }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n" ;
    }
};

int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

C++ OBJECT ORIENTED

C++ Polymorphism(다형성)

- 기본 클래스가 다른 두 클래스에 의해 파생된 다음 예를 살펴보자.

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

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;

   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;

   // call triangle area.
   shape->area();

   return 0;
}
```

```
Parent class area :
Parent class area :
```

C++ OBJECT ORIENTED

C++ Polymorphism(다형성)

- 잘못된 출력의 이유는 함수 area()의 호출이 기본 클래스에 정의된 버전으로 컴파일러에 의해 한 번 설정되기 때문이다. 이것을 함수 호출의 정적 해결 또는 정적 연결이라고 한다. 함수 호출은 프로그램이 실행되기 전에 고정된다. 이는 프로그램 컴파일 중에 area() 함수가 설정되기 때문에 초기 바인딩이라고도 한다.
- 그러나 이제 프로그램을 약간 수정하고 Shape 클래스의 area() 선언 앞에 virtual 키워드를 사용하여 다음과 같이 살펴보자.

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

C++ OBJECT ORIENTED

C++ Polymorphism(다형성) - Virtual Function

- 가상 함수는 virtual 키워드를 사용하여 선언된 기본 클래스의 함수이다. 파생 클래스의 다른 버전과 함께 기본 클래스에서 가상 함수를 정의하면 이 함수에 대한 정적 연결(static linkage)을 원하지 않는다는 신호가 컴파일러에 전달된다.
- 우리가 원하는 것은 호출되는 객체의 종류를 기반으로 프로그램의 주어진 지점에서 호출될 함수를 선택하는 것이다. 이러한 종류의 작업을 동적 연결(dynamic linkage) 또는 후기 바인딩(late binding)이라고 한다.

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

C++ OBJECT ORIENTED

C++ Polymorphism(다형성) – Pure Virtual Function

- 기본 클래스에 가상 함수를 포함하여 해당 클래스의 객체에 맞게 파생 클래스에서 재정의할 수 있지만 기본 클래스의 함수에 대해 의미 있는 정의를 제공할 수 없다.
- 기본 클래스의 가상 함수를 다음과 같이 변경할 수 있다.
- = 0은 함수에 본문이 없고 가상 함수를 순수 가상 함수(Pure Virtual Function)라고 부를 것임을 컴파일러에 알린다.

```
class Shape {
   protected:
      int width, height;

   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      // pure virtual function
      virtual int area() = 0;
};
```

C++ OBJECT ORIENTED

C++ Polymorphism(다형성) – abstract class(추상 클래스)

- C++에서는 하나 이상의 순수 가상 함수를 포함하는 클래스를 추상 클래스(abstract class)라고 한다.
- 이러한 추상 클래스는 객체 지향 프로그래밍에서 중요한 특징인 다형성을 가진 함수의 집합을 정의할 수 있게 해준다.
- 즉, 반드시 사용되어야 하는 멤버 함수를 추상 클래스에 순수 가상 함수로 선언해 놓으면, 이 클래스로부터 파생된 모든 클래스에서는 이 가상 함수를 반드시 재정의해야 한다.
- 추상 클래스는 동작이 정의되지 않은 순수 가상 함수를 포함하고 있으므로, 인스턴스를 생성할 수 없다.
- 따라서 추상 클래스는 먼저 상속을 통해 파생 클래스를 만들고, 만든 파생 클래스에서 순수 가상 함수를 모두 overriding하고 나서야 비로소 파생 클래스의 인스턴스를 생성할 수 있게 된다.
- 하지만 추상 클래스 타입의 포인터와 참조는 바로 사용할 수 있다.

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

class Animal
{
public:
    virtual ~Animal() {}    // 가상 소멸자의 선언
    virtual void Cry()=0;   // 순수 가상 함수의 선언
};

class Dog : public Animal
{
public:
    virtual void Cry() { cout << "멍멍!!" << endl; }
};

class Cat : public Animal
{
public:
    virtual void Cry() { cout << "야옹야옹!!" << endl; }
};

int main(void)
{
    Dog my_dog;
    my_dog.Cry();
    Cat my_cat;
    my_cat.Cry();
    return 0;
}
```

C++ OBJECT ORIENTED

C++ Polymorphism(다형성) – abstract class(추상 클래스)

- 추상 클래스의 용도 제한
- C++에서 추상 클래스는 다음과 같은 용도로는 사용할 수 없다.
  - 변수 또는 멤버 변수
  - 함수의 전달되는 인수 타입
  - 함수의 반환 타입
  - 명시적 타입 변환의 타입

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

class Animal
{
public:
    virtual ~Animal() {}    // 가상 소멸자의 선언
    virtual void Cry()=0;   // 순수 가상 함수의 선언
};

class Dog : public Animal
{
public:
    virtual void Cry() { cout << "멍멍!!" << endl; }
};

class Cat : public Animal
{
public:
    virtual void Cry() { cout << "야옹야옹!!" << endl; }
};

int main(void)
{
    Dog my_dog;
    my_dog.Cry();
    Cat my_cat;
    my_cat.Cry();
    return 0;
}
```

C++ OBJECT ORIENTED

C++ Polymorphism(다형성) – overriding

- 상속 overriding을 보자, 전에 배웠던 함수 overloading이 생각난다.
- overloading이 매개변수의 자료형이나 수가 다른 함수를 같은 이름으로 여러 번 중복 정의하는 것이라면
- overriding은 이미 있는 함수를 무시하고 새롭게 함수를 재정의하는 것이다. 더 자세히 말하자면, 이 오버라이딩(Overriding, 재정의)는 부모 클래스와 자식 클래스의 상속 관계에서, 부모 클래스에 이미 정의된 함수를 같은 이름으로 자식 클래스에서 재정의 하는 것이다.
- 이 때, 부모의 멤버 함수와 원형이 완전히 같아야 한다. 그리고 오버라이딩시 부모 클래스의 함수가 모두 가려진다.

```cpp
#include <iostream>

using namespace std;

class A {
public:
    void over() { cout << "A 클래스의 over 함수 호출!" << endl; }
};

class B : public A {
public:
    void over() { cout << "B 클래스의 over 함수 호출!" << endl; }
};

int main()
{
    B b;
    b.over();
    return 0;
}
```

C++ OBJECT ORIENTED

Data Abstraction in C++(Data 추상화)

- 데이터 추상화는 외부 세계에 필수적인 정보만 제공하고 배경 세부 정보를 숨기는 것, 즉 세부 정보를 제시하지 않고 프로그램에서 필요한 정보를 표현하는 것을 말한다.
- 데이터 추상화는 인터페이스와 구현의 분리에 의존하는 프로그래밍 및 디자인 기술이다.
- C++에서 클래스는 뛰어난 수준의 데이터 추상화를 제공한다.
- 객체의 기능을 사용하고 객체 데이터, 즉 클래스가 내부적으로 구현된 방법을 실제로 알지 못하는 상태를 조작할 수 있도록 외부 세계에 충분한 public method를 제공한다.
- C++에서는 클래스를 사용하여 고유한 ADT(Abstract Data Types;추상 데이터 유형)를 정의한다.

C++ OBJECT ORIENTED

Data Abstraction in C++(Data 추상화)

- 액세스 레이블(Access Labels)은 추상화를 시행한다.
- C++에서는 액세스 레이블을 사용하여 클래스에 대한 abstract interface(추상 인터페이스)를 정의한다.
- 클래스는 0개 또는 그 이상의 액세스 레이블을 포함할 수 있다.
  - Public label로 정의된 멤버는 프로그램의 모든 부분에 액세스할 수 있다. 형식의 데이터 추상화 보기는 해당 public member에 의해 정의된다.
  - Private label로 정의된 멤버는 클래스를 사용하는 코드에 액세스할 수 없다. private 섹션은 해당 유형을 사용하는 코드에서 구현을 숨긴다.

C++ OBJECT ORIENTED

Data Abstraction in C++(Data 추상화)

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

class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }

      // interface to outside world
      void addNum(int number) {
         total += number;
      }

      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};

int main() {
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

C++ OBJECT ORIENTED

Interfaces in C++ (Abstract Classes)

- Interface는 해당 클래스의 특정 구현을 커밋하지 않고 C++ 클래스의 동작 또는 기능을 설명한다.
- C++ Interface는 추상 클래스를 사용하여 구현되며 이러한 추상 클래스는 구현 세부 정보를 관련 데이터와 별도로 유지하는 개념인 데이터 추상화와 혼동되어서는 안 된다.
- 클래스는 최소한 하나의 함수를 순수 가상 함수로 선언함으로써 추상화된다.

C++ OBJECT ORIENTED

Interfaces in C++ (Abstract Classes)

- Abstract class(추상 클래스(종종 ABC라고 함))의 목적은 다른 클래스가 상속할 수 있는 적절한 기본 클래스를 제공하는 것이다.
- 추상 클래스는 객체(object)를 인스턴스화(instantiate)하는 데 사용할 수 없으며 Interface 역할만 한다. 추상 클래스의 객체를 인스턴스화하려고 하면 컴파일 오류가 발생한다.
- 따라서 ABC(Abstract class)의 하위 클래스를 인스턴스화해야 하는 경우 각 virtual function을 구현해야 하며, 이는 Abstract class에서 선언한 인터페이스를 지원한다는 것을 의미한다.
- 파생 클래스에서 순수 가상 함수를 재정의하지 못한 다음 해당 클래스의 개체를 인스턴스화하려고 하면 컴파일 오류가 발생한다.
- 객체를 인스턴스화하는 데 사용할 수 있는 클래스를 concrete class(구체 클래스)라고 한다.

C++ OBJECT ORIENTED

Interfaces in C++ (Abstract Classes)

- 부모 클래스가 getArea()라는 함수를 구현하기 위해 기본 클래스에 대한 인터페이스를 제공하는 예제를 살펴 보자.

```cpp
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};

int main(void) {
   Rectangle Rect;
   Triangle  Tri;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```