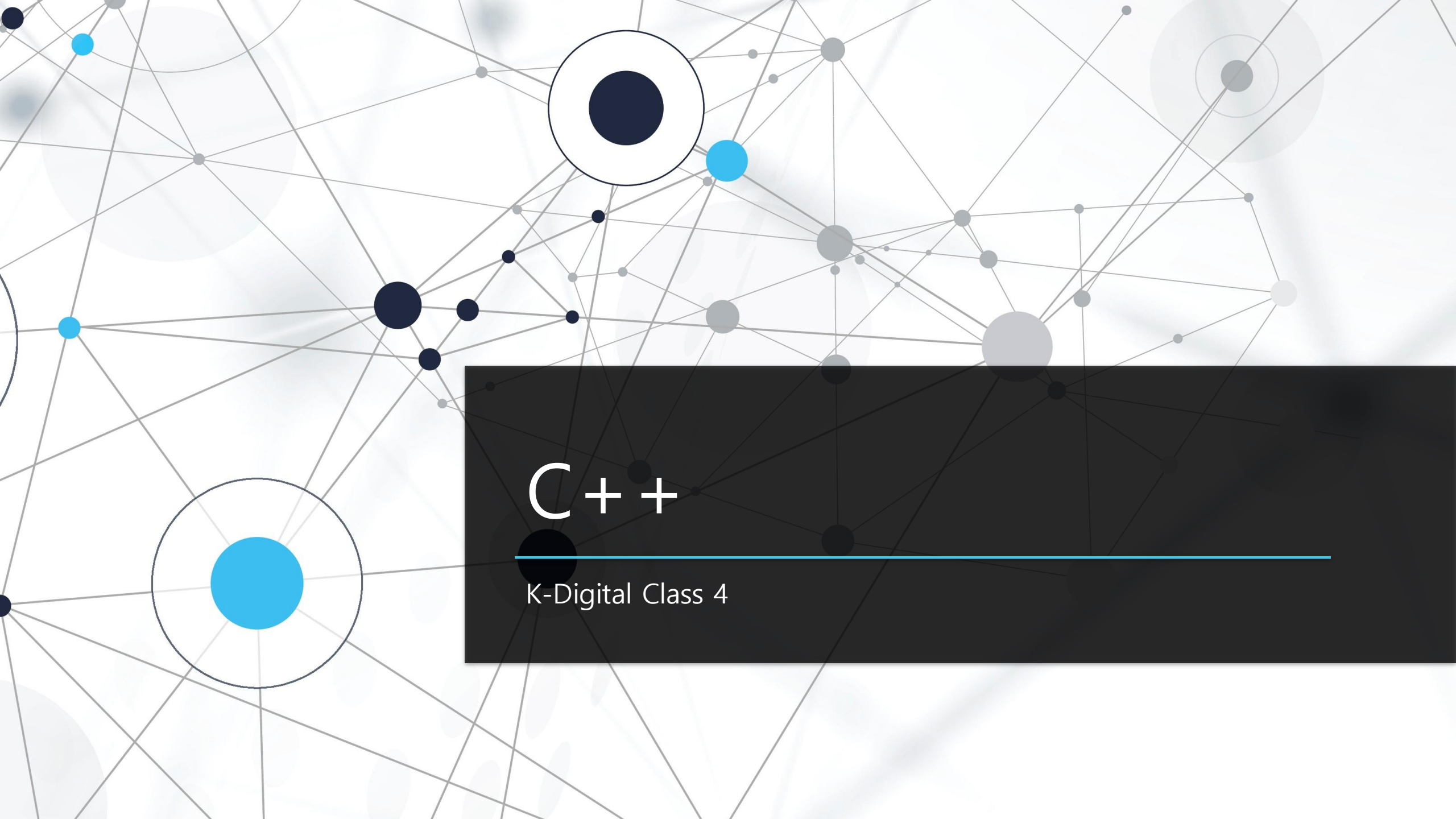

C++

K-Digital Class 4

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