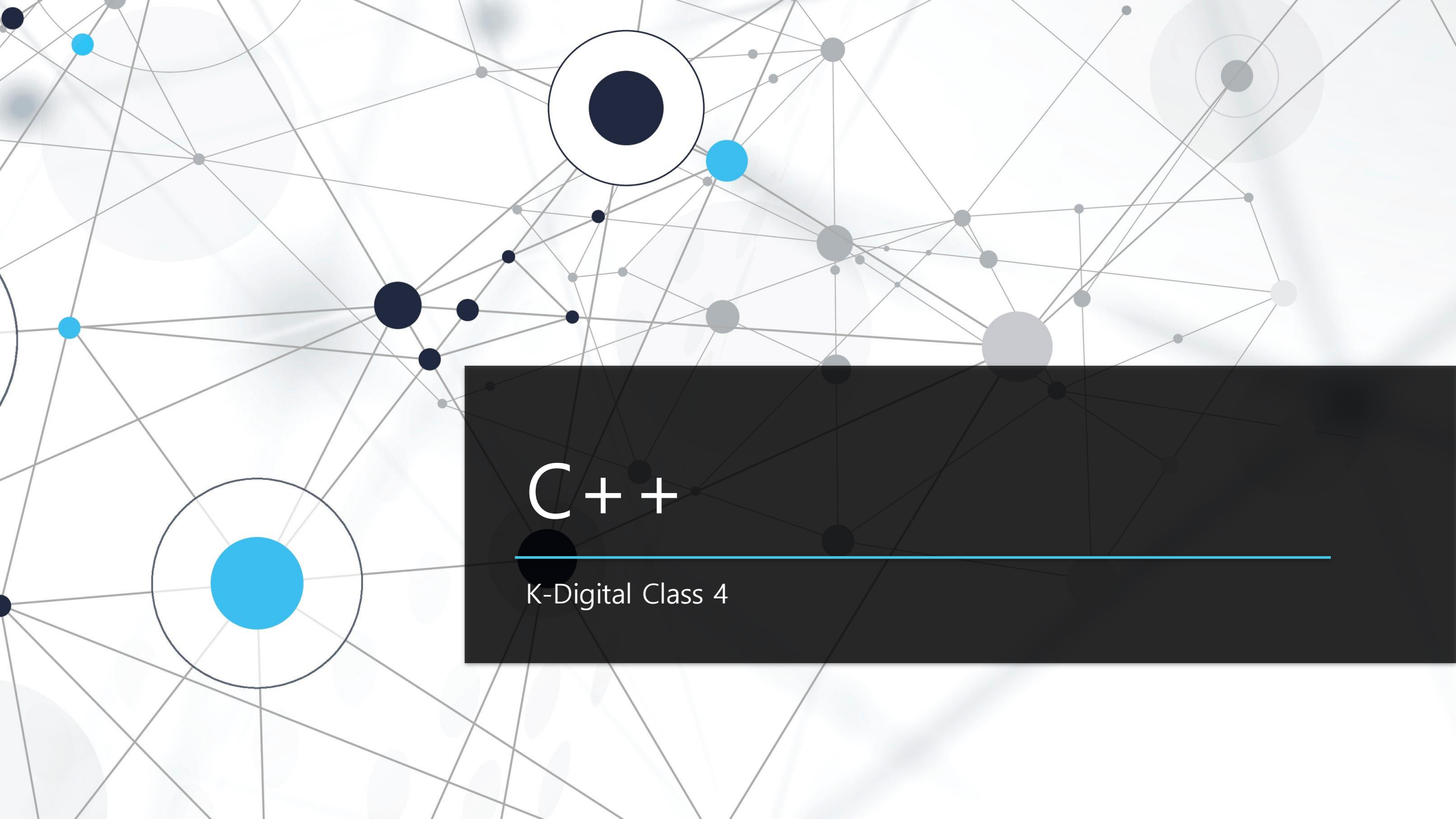


C++

K-Digital Class 4

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

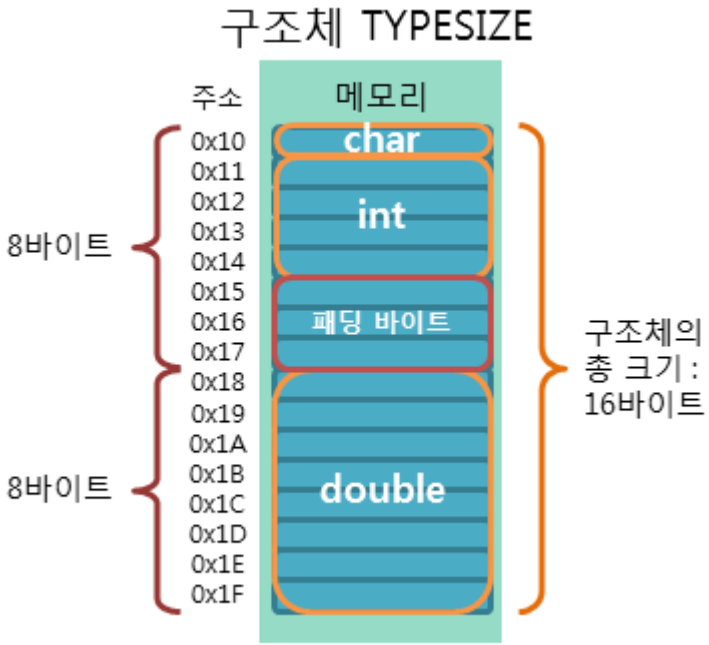
- 위의 예제에서 구조체 멤버 변수의 크기는 각각 1, 4, 8바이트이다.
- 하지만 구조체의 크기는 멤버 변수들의 크기 총합인 13바이트가 아니라 16바이트가 된다.
- 구조체를 메모리에 할당할 때 컴파일러는 프로그램의 속도 향상을 위해 바이트 패딩(byte padding)이라는 규칙을 이용한다.
- 구조체는 다양한 크기의 타입을 멤버 변수로 가질 수 있는 타입이다.
- 하지만 컴파일러는 메모리의 접근을 쉽게 하려고 크기가 가장 큰 멤버 변수를 기준으로 모든 멤버 변수의 메모리 크기를 맞추게 된다.
- 이것을 바이트 패딩이라고 하며,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

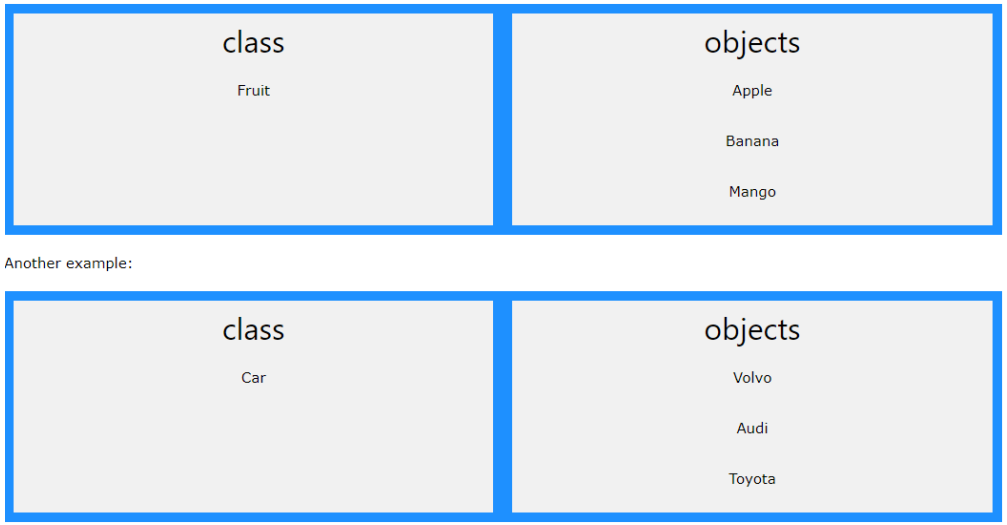
- 객체 지향 프로그래밍은 절차적 프로그래밍에 비해 몇 가지 장점이 있다.
  - OOP는 더 빠르고 쉽게 실행할 수 있다.
  - OOP는 프로그램에 대한 명확한 구조를 제공한다.
  - OOP는 C++ 코드 DRY ("Don't Repeat Yourself")를 유지하는 데 도움이 되며 코드를 유지 관리, 수정 및 디버그하기 쉽게 만든다.
  - OOP를 사용하면 더 적은 코드와 더 짧은 개발 시간으로 완전히 재사용 가능한 application을 만들 수 있다.

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;
}
```