1. Hello world로 본 C++

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello world! \n";

    return 0;
}
```

- std는 '네임스페이스(Namespace)' 라고 하며 개념 상 '소속'으로 생각하면 됨.
- :: 은 '범위 지정 연산자' 혹은 '스코프 설정 연산자(Scope resolution operator)' 라고 함.
- cout 은 콘솔 출력을 담당하는 객체
- << 은 '연산자 함수'

▶ std에 속한 cout 객체에 "Hello world! \n" 문자열을 '넘겨(<<)' 문자열을 화면에 출력하라!

2. 인스턴스(instance)

```cpp
int a;
string strdata;
```

- a는 int 자료형에 대한 인스턴스
- strdata는 string 자료형에 대한 인스턴스
- C++에서는 '변수'라는 표현보다 '인스턴스' 라는 표현에 익숙해져야 함.
- 객체지향 프로그래밍환경에서 모든 것을 객체로 표현하고 객체의 형식을 갖는 변수를 인스턴스라고 함.

3. std::cout, std::cin

- 실습 : cin 객체로 입력받고 cout 객체로 출력해보기

```
나이를 입력하세요 : 20
직업을 입력하세요 : 학생
이름을 입력하세요 : 홍길동
당신의 이름은 홍길동이고, 나이는 20살이며, 직업은 학생입니다.
```

```cpp
#include <iostream>
#include <string>
int main()
{
    // 위의 예시처럼 입력받고 출력하는 예제 프로그램을 작성해서 실행시켜보세요~!
    // 문자열 입력은 std::string 객체를 이용
    // std::string strName;
    // std::cin >> strName; 으로 입력 가능
    return 0;
}
```

4. 자료형

- 자료형 : 일정 크기의 메모리에 저장된 정보를 해석하는 방법
- 기본 자료형은 C와 다르지 않음
- C++11 표준에서 추가된 것 확인

자료형	설명	
long long	64비트 정수	
char16_t	16비트 문자 (ex. char16_t ch = u'A';)	유니코드 처리를 위한 자료형
char32_t	32비트 문자 (ex. char32_t ch = u'A';)	
auto	컴파일러가 자동으로 형식을 규정하는 자료형(ex. auto a = 10;)	
decltype(expr)	expr과 동일한 자료형(ex. int x=10; decltype(x)y = 20;)	

■ 변수 선언 및 정의

```
int num = 10;    // C 스타일 선언 및 정의
int num2(10);    // C++ 스타일 선언 및 정의
int num3(num2);  // 이것도 가능
```

■ auto 예약어

- C++에서 초기값의 형식에 맞춰 선언하는 인스턴스의 형식이 '자동'으로 결정!

```
auto nData = 10;        // nData 형식은 int
auto strName = "Tom";   // strName 형식은 const char*
auto ch = 'A';          // ch 형식은 char
```

5. 메모리 동적할당

- new : 메모리 동적 할당 연산자
- delete : 메모리 해제 연산자

```
int* pData = new int;
delete pData;
```

```
char* mystr = new char[10];
delete[] mystr;
```

```
#include <iostream>
int main()
{
    // 인스터스만 동적으로 생성하는 경우
    int* pData = new int;

    // 초기값을 기술하는 경우
    int* pNewData = new int(10);

    *pData = 5;
    std::cout << *pData << std::endl;
    std::cout << *pNewData << std::endl;

    delete pData;
    delete pNewData;

    return 0;
}
```

```
#include <iostream>
int main()
{
    // 객체를 배열 형태로 동적 생성한다.
    int* arr = new int[5];

    for (int i = 0; i < 5; i++)
        arr[i] = (i + 1) * 10;

    for (int i = 0; i < 5; i++)
        std::cout << arr[i] << std::endl;

    // 배열 형태로 생성한 대상은
    // 반드시 배열 형태를 통해 삭제한다!
    delete[] arr;

    return 0;
}
```

6. 참조자 형식

■ 참조자 변수 선언과 정의

- C에 없는 형식으로 포인터와 구조적으로 비슷
- 선언과 동시에 반드시 초기화 해야함 (형식 &이름 = 원본;)

```
int *pNum = &3; // 상수에 포인터 선언 불가능
int &rNum = 3;  // 상수에 대한 참조는 불가능 ( 가능하게 하려면? )
int &rNum2;     // 참조 원본이 없으므로 불가능
```

- 실습예제

```
#include <iostream>
int main()
{
    int nData = 10;

    // nData 인스턴스에 대한 참조자 선언
    int& ref = nData;

    // 참조자의 값을 변경하면 원본도 변경된다!
    ref = 20;

    std::cout << nData << std::endl;

    // 포인터를 쓰는 것과 비슷하다.
    int* pnData = &nData;
    *pnData = 30;
    std::cout << nData << std::endl;

    return 0;
}
```

1. 실행결과 확인
2. nData 주소와 ref 주소 확인
3. nData 와 ref 관계 생각해보기!

참조자는 겉으로 보기에는
전혀 포인터로 보이지 않는다!

■ r-value 참조

- C++11 에 새로 등장한 문법 (형식 &&이름 = 원본; > **int &&rdata = 3;**)
- r-value - 대입 연산자의 두 피연산자 중 오른쪽에 위치한 것으로 일반적인 변수와 상수 모두 해당
  - 연산에 따라 생성된 임시객체! (나중에 추가로 배울 내용)

```
#include <iostream>
int TestFunc(int nParam)
{
    int nResult = nParam * 2;
    return nResult;
}
int main()
{
    int nInput = 0;
    std::cout << "Input number : ";
    std::cin >> nInput;

    // 산술 연산으로 만들어진 임시 객체에 대한 r-value 참조
    int&& rdata = nInput + 5;
    std::cout << rdata << std::endl;

    // 함수 반환으로 만들어진 임시 객체에 대한 r-value 참조
    int&& result = TestFunc(10);

    // 값을 변경할 수 있다.
    result += 10;
    std::cout << result << std::endl;

    return 0;
}
```

1. 실행결과 확인
2. r-value, l-value 공부해보기

7. 범위기반 for문

- C++11 에 새로 등장한 문법
- 반복 횟수는 배열 요소 개수에 맞춰 자동으로 결정

```
    for (auto 요소변수 : 배열이름)
        반복 구문;
```

```
#include <iostream>
int main()
{
    int nList[5] = { 10, 20, 30, 40, 50 };

    // 전형적인 C 스타일 반복문
    for (int i = 0; i < 5; i++)
        std::cout << nList[i] << " ";

    std::cout << std::endl;

    // 범위기반 C++11 스타일 반복문
    // 각 요소의 값을 n에 복사한다.
    for (auto n : nList)
        std::cout << n << " ";

    std::cout << std::endl;

    return 0;
}
```

```
정리
1. cout, cin 객체로 입출력
2. auto 예약어 (의미, 사용방법 등)
3. 동적 메모리 할당, 해제 ( new, delete )
4. 참조형식 & 상수형 참조도 생각해보기
5. 범위기반 for 문
```

8. 디폴트 매개변수

- 매개변수를 디폴트로 초기값 지정
- 매개변수의 디폴트 값을 '선언'한 함수는 호출자 코드에서 실인수를 생략한 채 호출 가능

```
#include <iostream>
// nParam 매개변수의 디폴트 값은 10이다.
int TestFunc(int nParam = 10)
{
    return nParam;
}
int main()
{
    // 호출자가 실인수를 기술하지 않았으므로 디폴트 값을 적용
    std::cout << TestFunc() << std::endl;

    // 호출자가 실인수를 확정했으므로 디폴트 값을 무시
    std::cout << TestFunc(30) << std::endl;
    return 0;
}
```

- 함수의 선언과 정의를 나눌 경우 선언 부분에 디폴트 값을 기술해야 함

```
#include <iostream>
// 함수 선언 부분에서 디폴트 값 기술
int TestFunc(int nParam = 10);
int main()
{
    std::cout << TestFunc(40) << std::endl;
    return 0;
}
// 함수 정의 부분
int TestFunc(int nParam)
{
    return nParam;
}
```

- 매개 변수가 두 개 일때의 디폴트 값

```
#include <iostream>
int TestFunc(int nParam, int nParam2 = 2)
{
    return nParam * nParam2;
}
int main()
{
    std::cout << TestFunc(10) << std::endl;
    std::cout << TestFunc(10, 5) << std::endl;
    return 0;
}
```

● 디폴트 매개변수 사용시 기억해둘것!

✓ 피호출자 함수 매개변수의 디폴트 값은 반드시 오른쪽 매개변수부터 기술해야 함.

✓ 매개변수가 여러개일 때 왼쪽 첫 번째 매개변수의 디폴트 값을 기술하려면 나머지 오른쪽 '모든' 매개변수에 대한 디폴트 값을 기술해야 함. 절대로 중간에 빼먹으면 안된다.

✓ 호출자 함수가 피호출자 함수 매개변수의 실인수를 기술하면 이는 왼쪽부터 짝을 맞추어 적용되며, 짝이 맞지 않는 매개변수는 디폴트 값을 적용한다.

● 디폴드 값이 없는 매개변수에는 호출자 함수에 반드시 실인수를 기술해야 함

```
#include <iostream>
int TestFunc(int nParam, int nParam2 = 2)
{
    return nParam * nParam2;
}
int main()
{
    std::cout << TestFunc() << std::endl; // error 발생 nParam 매개변수 있어야 함!!
    return 0;
}
```

```
error C2660: 'TestFunc': 함수는 0개의 인수를 사용하지 않습니다.
```

● 디폴드 값은 오른쪽 매개변수부터 기술해야 함

```
#include <iostream>
int TestFunc(int nParam = 5, int nParam2)
{
    return nParam * nParam2;
}
int main()
{
    std::cout << TestFunc(10) << std::endl;
    return 0;
}
```

```
// error C2548: 'TestFunc': 매개 변수 2의 기본 인수가 없습니다.
```

● 중간에 위치한 매개변수에 디폴트 값을 생략할 수 없음

```
#include <iostream>
int TestFunc(int nParam1 = 5, int nParam2, int nParam3 = 10)
{
    return nParam1 * nParam2 * nParam3;
}
int main()
{
    std::cout << TestFunc(10, 20) << std::endl;
    return 0;
}
```

```
error C2548: 'TestFunc': 매개 변수 2의 기본 인수가 없습니다.
```

9. 함수 다중정의(Overloading)

- C++에서 다중정의는 하나(함수이름, 변수이름 등)가 여러의미를 동시에 갖는 것
- C 에서는 이름이 같은 함수 존재할 수 없음
- C++ 매개변수 구성이 달라지거나 어떤 식으로든 함수 원형이 달라지면 이름이 같더라도 OK!
- C++ 는 함수의 '다형성'을 지원

```
#include <iostream>
int Add(int a, int b, int c)
{
    std::cout << "Add(int, int, int): ";
    return a + b + c;
}
int Add(int a, int b)
{
    std::cout << "Add(int, int): ";
    return a + b;
}
double Add(double a, double b)
{
    std::cout << "Add(double, double): ";
    return a + b;
}
int main()
{
    std::cout << Add(3, 4) << std::endl;
    std::cout << Add(3, 4, 5) << std::endl;
    std::cout << Add(3.3, 4.4) << std::endl;

    return 0;
}
```

- 함수원형의 구성을 살펴보면 크게 네가지(반환형식, 호출규칙, 함수이름, 매개변수)에서 다중정의에 영향을 주는 것은 '매개변수' 뿐이다.

```
반환형식 호출규칙 함수이름(매개변수, 매개변수, ...);
```

- 문법에 맞지 않는 예1: 반환형식만 다른경우

```
int Add(int a, int b);
double Add(int a, int b);
```

- 문법에 맞지 않는 예2 : 호출규칙만 다른 경우

```
int __cdecl Add(int a, int b);
int __stdcall Add(int a, int b);
```

■ 다중정의와 모호성

- 디폴트 매개변수와 다중정의가 조함되면 매우 강력한 모호성이 발생할 수 있음

```
01  #include <iostream>
02  void TestFunc(int a)
03  {
04      std::cout << "TestFunc(int)" << std::endl;
05  }
06  void TestFunc(int a, int b = 10)
07  {
08      std::cout << "TestFunc(int, int)" << std::endl;
09  }
10  int main()
11  {
12      TestFunc(5);
13
14      return 0;
15  }
```

- 위의 소스를 실행해보기전에 먼저 생각해볼 것

✓ 컴파일할 수 있는가? 즉, 컴파일 오류는 없는가?

✓ 오류가 발생한다면 몇 번 행에서 발생하는가?

✓ TestFunc() 함수의 다중 정의가 문제인가? 아니면 호출하는 쪽이 문제인가?

■ 함수 템플릿

- 함수를 다중 정의하는 이유는 사용자의 편의성과 확장성을 얻을 수 있음
- 코드 제작자는 같은 일을 여러번 반복해야 함. 같은 일을 하는 코드가 다중 정의된 함수 여러 개로 존재 -> (유지보수 측면에서 심각한 문제)
- 가급적이면 함수 다중 정의보다 '함수 템플릿(Function template)' 사용을 권장

```
template <typename T>
반환형식 함수이름(매개변수)
{
}
```

```
#include <iostream>
template <typename T>
T TestFunc(T a)
{
    std::cout << "매개변수 a : " << a << std::endl;
    return a;
}
int main()
{
    std::cout << "int \t" << TestFunc(3) << std::endl;
    std::cout << "double \t" << TestFunc(3.3) << std::endl;
    std::cout << "char \t" << TestFunc('A') << std::endl;
    std::cout << "char* \t" << TestFunc("TestString") << std::endl;

    return 0;
}
```

- 함수 템플릿으로 만든 Add() 함수

```
#include <iostream>
template <typename T>
T Add(T a, T b)
{
    return a+b;
}
int main()
{
    std::cout << Add(3, 4) << std::endl;
    std::cout << Add(3.3, 4.4) << std::endl;

    return 0;
}
```

10. 인라인 함수

- 함수를 호출하면 내부적으로 여러 연산들(매개변수 복사, 스택 조정, 제어 이동등)의 오버헤드 발생
- 매크로 : 함수가 아님(매개변수에 형식 지정할 수 없음)
- 인라인 함수 : 매크로의 장점과 함수의 장점만 모아놓은 것

```
#include <iostream>
#define ADD(a, b) ( (a) + (b) )
int Add(int a, int b)
{
    return a + b;
}
inline int AddNew(int a, int b)
{
    return a + b;
}
int main()
{
    int a, b;
    std::cin >> a >> b;

    std::cout << "ADD() : " << ADD(a, b) << std::endl;
    std::cout << "Add() : " << Add(a, b) << std::endl;
    std::cout << "AddNew() : " << AddNew(a, b) << std::endl;

    return 0;
}
```

- inline 예약어를 빼면 기존의 함수와 다를 바 없음
- 문법적으로 완벽한 함수
- 컴파일러 최적화

11. 네임스페이스(Namespace)

- C++ 가 지원하는 각종 요소들(변수, 함수, 클래스 등)을 한 범주로 묶어주기 위한 문법
- 의미상 소속, 구역
- namespace 선언방법

```
namespace 이름
{
    // namespace 시작
    ......
    // namepace 끝
}
```

- namespace 블록 내부에 선언하는 변수나 함수들은 모두 명시한 '이름'에 속하게 된다.

```
01  #include <iostream>
02  namespace TEST
03  {
04      int g_nData = 100;
05      void TestFunc()
06      {
07          std::cout << "TEST:: TestFunc()" << std::endl;
08      }
09  }
10  int main()
11  {
12      TEST::TestFunc();
13      std::cout << TEST::g_nData << std::endl;
14      return 0;
15  }
```

- 12번 행을 보면 namespace 가 존재할 경우 식별자 앞에 범위지정연산자(::)을 이용해 namespace를 기술할 수 있음.
- main() 함수는 Global namespace 에 속함.
- cout, TestFunc(), main() 등은 각자 속한 namespace 가 모두 다름.

■ using 선언

- 프로그램 내부에서 앞으로 자주 사용해야 하는 namespace가 있다면 모든 식별자 앞에 이를 기술할 것이 아니라 using 예약어를 선언한 후 namespace를 생략 가능

```
using namespace 네임스페이스이름;
```

```
#include <iostream>
// std 네임스페이스를 using 예약어로 선언한다.
using namespace std;

namespace TEST
{
    int g_nData = 100;

    void TestFunc()
    {
        // cout , endl 에 대해서 범위를 지정하지 않아도 된다. ( using 선언했기 때문)
        cout << "TEST:: TestFunc()" << endl;
    }
}
// TEST 네임스페이스를 using 예약어로 선언한다.
using namespace TEST;
int main()
{
    // TestFunc()나 g_nData 에 대해서 범위를 지정하지 않아도 된다. ( using 선언했기 때문)
    TestFunc();
    cout << g_nData << endl;
    return 0;
}
```

■ namespace 중첩

- 네임스페이스 안에 또 다른 네임스페이스가 속할 수 있음.

```
#include <iostream>
using namespace std;
namespace TEST
{
    int g_nData = 100;
    namespace DEV
    {
        int g_nData = 200;
        namespace WIN
        {
            int g_nData = 300;
        }
    }
}
int main()
{
    cout << TEST::g_nData << endl;
    cout << TEST::DEV::g_nData << endl;
    cout << TEST::DEV::WIN::g_nData << endl;
    return 0;
}
```

- int g_nData 변수는 이름은 같아도 전혀 다른 세 개의 전역 변수
- 접근할 때는 정확히 namespace 를 명시해야 함
- cout << g_nData << endl;을 실행 시 컴파일 오류 발생

■ namespace와 다중정의

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

// 전역(개념상 무소속)
void TestFunc() {
    cout << "::TestFunc()" << endl;
}

namespace TEST
{
    // TEST namespace 소속
    void TestFunc()
    {
        cout << "TEST::TestFunc()" << endl;
    }
}
namespace MYDATA
{
    // MYDATA namespace 소속
    void TestFunc()
    {
        cout << "MYDATA::TestFunc()" << endl;
    }
}
int main()
{
    TestFunc();     // 묵시적 전역
    ::TestFunc();   // 명시적 전역
    TEST::TestFunc();
    MYDATA::TestFunc();
    return 0;
}
```

- TestFunc() 함수는 세 번 정의 되는데 각각 속한 namespace가 다름.
- TEST, MYDATA namespace에 using 선언을 하면 어떻게 될까?

```
using namespace TEST;
using namespace MYDATA;
```

12. 식별자 검색순서

- 식별자가 선언된 위치를 검색하는 순서

전역 함수인 경우

1. 현재 블록 범위
2. 현재 블록 범위를 포함하고 있는 상위 블록범위(최대 적용 범위는 함수 몸체까지)
3. 가장 최근에 선언된 전역 변수나 함수
4. using 선언된 namespace 혹은 global namespace. 단, 두 곳에 동일한 식별자가 존재할 겨우 컴파일 오류 발생

class method 경우

1. 현재 블록 범위
2. 현재 블록 범위를 포함하고 있는 상위 블록범위(최대 적용 범위는 함수 몸체까지)
3. class member
4. 부모 class member
5. 가장 최근에 선언된 전역 변수나 함수
6. 호출자 코드가 속한 namespace의 상위 namespace
7. using 선언된 namespace 혹은 global namespace. 단, 두 곳에 동일한 식별자가 존재할 겨우 컴파일 오류 발생

- 현재 블록 범위({ } 구간)

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

int nData(20);

int main()
{
    int nData(30);
    cout << nData << endl;
    return 0;
}
```

- 상위 블록 범위

```
#include <iostream>
using namespace std;
int main()
{
    int nInput(0);
    cout << "1 이상의 자연수를 입력하세요 : " << endl;
    cin >> nInput;

    if (nInput % 2) {
        cout << nInput << "은 홀수입니다. " << endl;
    }
    else {
        int nInput(nInput);
        cout << nInput << "은 짝수입니다. " << endl;
    }
    return 0;
}
```

- 가장 최근에 선언된 전역변수(실행결과 비교)

```
#include <iostream>
using namespace std;
int nData(200);

namespace TEST
{
    int nData(100);
    void TestFunc()
    {
        cout << nData << endl;
    }
}
int main()
{
    TEST::TestFunc();
    return 0;
}
```

```
#include <iostream>
using namespace std;
int nData(200);

namespace TEST
{
    void TestFunc()
    {
        cout << nData << endl;
    }
    int nData(100);
}
int main()
{
    TEST::TestFunc();
    return 0;
}
```

- using 선언과 전역변수
- 아래 실행결과가 어떻게 될지 코드 보고 예측해보기

```
#include <iostream>
using namespace std;
int nData(200);

namespace TEST
{
    int nData(100);
}
int main()
{
    cout << nData << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int nData(200);

namespace TEST
{
    int nData(100);
}

using namespace TEST;

int main()
{
    cout << nData << endl;
    return 0;
}
```

정리문제

1. 다음 두 함수원형에서 잘못된 점은 무엇인가요?

```
int TestFunc(int nParam1 = 5, int nParam2, int nParam3 = 10);
int TestFunc(int nParam1 = 5, int nParam2);
```

2. 다음 두 함수는 문법적으로 문제가 없습니다. 하지만 호출하는 코드에서는 문제가 발생할 수 있습니다. 어떤 문제인가요?

```
void TestFunc(int a)
{
    std::cout << "TestFunc(int)" << std::endl;
}
void TestFunc(int a, int b = 10)
{
    std::cout << "TestFunc(int, int)" << std::endl;
}
```

3. 함수를 다중정의하는 것보다는 함수 템플릿이 더 좋은 코드가 될 가능성이 높습니다. 이유는?

4. inline 함수와 매크로의 공통된 장점은 무엇인가요?

5. namespace를 매번 작성하기 싫다면 미리 ( ) 선언을 하는 것이 좋습니다. 괄호 속에 들어갈 알맞은 말은?