Chapter 2

디폴트 매개변수 : 기본값을 정해서 함수 이름만 쓰면 함수를 실행

다중 정의 : 이름은 같은데 내 맘대로 자료형을 바꿔도 된다.

인라인 함수 : 함수와 매크로의 장점을 모은 것

네임스페이스 : 내가 원하는 함수, 변수 등에 소속감을 심어줍니다.

디폴트 매개변수

```
/* 디폴트 매개변수 사용 */
#include "pch.h"
#include <iostream>

// nParam 매개변수의 디폴트 값은 10이다.
int TestFunc(int nParam = 10) // == int TestFunc(int = 10); int TestFunc(int nParam) { }
{
    return nParam;
}

int main()
{
    // 호출자가 실인수를 기술하지 않았으므로 디폴트 값을 적용한다.
    std::cout << TestFunc() << std::endl;

    // 호출자가 실인수를 확정했으므로 디폴트 값을 무시한다.
    std::cout << TestFunc(20) << std::endl;

    return 0;
}
```

매개변수의 디폴트 값을 '선언'한 함수는 호출자 코드에서 실인수를 생략한 채 호출 가능

호출자의 코드만 보고 함수 원형을 확정하면 안 된다.

```
/* 매개변수가 두 개일 때의 디폴트 값 */
#include "pch.h"
#include <iostream>

int TestFunc(int nParam1, int nParam2 = 2)
{
    return nParam1 * nParam2;
}

int main()
{
    std::cout << TestFunc(10) << std::endl;
    std::cout << TestFunc(10, 5) << std::endl;
```

```
    return 0;
}
```

주의사항

매개변수의 디폴트의 값은 반드시 오른쪽부터 기술 (중간에 빼먹지 않기)

웬만하면 쓰지 않기 (함수 만든 자와 사용자가 다를 때는 쓰는 것이 좋다)

‘미래’의 유지보수 문제에 대응할 수 있도록 ‘현재’ 코드를 작성

함수 다중 정의

- 하나가 여러 의미를 동시에 갖는 것

```
/*  Add() 함수의 다중 정의 */
#include "pch.h"
#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;
}
```

매개변수가 달라야 한다

잘못된 코드

```
/* 디폴트 매개변수와 다중 정의가 조합되었을 때의 모호성 */
#include "pch.h"
#include <iostream>

void TestFunc(int a)
{
    std::cout << "TestFunc(int)" << std::endl;
}

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

int main()
{
    TestFunc(5);

    return 0;
}
```

호출이 모호해진다

함수 템플릿

```
/* 템플릿 함수 */
```

```
/* 함수 템플릿으로 만든 Add() 함수 */
```

```
#include "pch.h"
#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;
}
```

```
#include "pch.h"
#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;
}
```

안정적인 구조

인라인 함수

- 함수와 매크로의 장점만 한 군데 모아놓은 것

```
/* 인라인 함수 */
#include "pch.h"
#include <cstdio>

#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;
    scanf_s("%d%d", &a, &b);

    printf("ADD(): %d", ADD(a, b));
    printf("Add(): %d", Add(a, b));
    printf("AddNew(): %d", AddNew(a, b));

    return 0;
}
```

네임스페이스

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

```
/* 네임스페이스 선언 및 정의 */
#include "pch.h"
#include <iostream>

namespace TEST
{
    int g_nData = 100;

    void TestFunc(void)
    {
        std::cout << "TEST::TestFunc()" << std::endl;
    }
}

int main()
{
    TEST::TestFunc();
    std::cout << TEST::g_nData << std::endl;

    return 0;
}
```

프로젝트 시 이름이 잘 안 겹치게 한다

```
/* using 선언 */
#include "pch.h"
#include <iostream>

// std 네임스페이스를 using 예약어로 선언한다.
using namespace std;

namespace TEST
{
    int g_nData = 100;

    void TestFunc(void)
    {
        // cout에 대해서 범위를 지정하지 않아도 상관없다.
        cout << "TEST::TestFunc()" << endl;
    }
}

// TEST 네임스페이스에 using 선언을 한다.
using namespace TEST;

int main()
{
    // TestFunc()나 g_nData에도 범위 지정을 할 필요가 없다.
    TestFunc();
    cout << g_nData << endl;

    return 0;
}
```

네임스페이스의 중첩

```
/* 네임스페이스의 중첩 */
#include "pch.h"
#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;
}
```

```
/* 네임스페이스를 포함한 다중 정의 */
#include "pch.h"
```

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

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

namespace TEST
{
    // TEST 네임스페이스 소속
    void TestFunc(void) {
        cout << "TEST::TestFunc()" << endl;
    }
}

namespace MYDATA
{
    // MYDATA 네임스페이스 소속
    void TestFunc(void) {
        cout << "DATA::TestFunc()" << endl;
    }
}

int main()
{
    TestFunc();     // 묵시적 전역
    ::TestFunc();   // 명시적 전역
    TEST::TestFunc();
    MYDATA::TestFunc();

    return 0;
}
```

네임스페이스를 기술함으로써 각각을 구별해 호출

모호성을 제거하기 위해 네임스페이스를 구체적으로 명시해야 한다.

식별자 검색 순서

- 식별자가 선언된 위치를 검색하는 순서

전역 함수인 경우

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

클래스 매서드인 경우

1. 현재 블록 범위
2. 현재 블록 범위를 포함하고 있는 상위 블록 범위(최대 적용 범위는 함수 몸체까지)
3. 클래스의 멤버

4. 부모 클래스의 멤버
5. 가장 최근에 선언된 전역 변수나 함수
6. 호출자 코드가 속한 네임스페이스의 상위 네임스페이스
7. using 선언된 네임스페이스 혹은 전역 네임스페이스. 단, 두 곳에 동일한 식별자가 존재할 경우 컴파일 오류 발생!

```
/* 식별자에 접근하는 코드가 속한 블록 범위 */
#include "pch.h"
#include <iostream>
using namespace std;

int nData(20);

int main(int argc, char* argv[])
{
    int nData(10);

    cout << nData << endl;
    cout << argc << endl;

    return 0;
}
```

```
/* 범위 검색의 확장 */
#include "pch.h"
#include <iostream>
using namespace std;

int main()
{
    int nInput = 0;
    cout << "11 이상의 정수를 입력하세요" << endl;
    cin >> nInput;

    if (nInput > 10)
    {
        cout << nInput << endl;
    }

    else
        cout << "Error" << endl;

    return 0;
}
```

```
/* 네임스페이스와 전역 변수의 검색 우선권 */
#include "pch.h"
#include <iostream>
using namespace std;

int nData = 200;

namespace TEST
{
    int nData = 100;
    void TestFunc(void)
    {
        cout << nData << endl;
    }
}

int main()
{
    TEST::TestFunc();

    return 0;
}
```

변형

```
int nData = 200;

namespace TEST
{
    void TestFunc(void)
    {
        cout << nData << endl;
    }
  int nData = 100;
}
```

출력 : 200

```
namespace TEST
{
    void TestFunc(void)
    {
        cout << nData << endl;
    }
  int nData = 100;
}
```

컴파일 오류

```
/* using namespace 선언을 적용하기 전 */
#include "pch.h"
#include <iostream>
using namespace std;

int nData = 100;

namespace TEST
{
    int nData = 200;
}

int main()
{
    cout << nData << endl;

    return 0;
}
```

출력 : 100

```
/* TEST 네임스페이스에 using 선언 추가 */
#include "pch.h"
#include <iostream>
using namespace std;

int nData = 100;

namespace TEST
{
    int nData = 200;
}

using namespace TEST;

int main()
{
    cout << nData << endl;

    return 0;
}
```

컴파일 오류 - 모호함

문제

잘못된 코드를 고치세요

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

- 디폴트 값은 오른쪽부터 차례대로 줘야한다

실행이 되지않은 이유를 찾아 고치세요

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

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

- 만약 TestFunc(5)를 한다면 모호해짐

다중정의 보다 함수 템플릿이 더 좋은 이유

inline 함수와 매크로의 공통된 장점

모호성을 줄여주고 코드의 길이가 짧아짐

네임스페이스를 미리 선언하는 방법

using

내부적으로 많은 연산을 줄여줌 - 성능 향상

다음 코드의 실행 결과를 작성하세요.

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

int nData = 200;

namespace TEST
{
    int nData = 100;
    void TestFunc(void)
    {
        cout << nData << endl;
    }
}

int main()
{
    TEST::TestFunc();

    return 0;
}
```

출력 : 100