C++ 기초

printf와 scanf

- **출력의 기본 형태 : 과거 스타일!**
  - **iostream.h 헤더 파일의 포함**

```
cout << 출력 대상;
```

```
cout<<출력 대상1<<출력 대상2<<출력 대상3;
```

```
cout<<1<<'a'<<"String"<<endl;
```

- 출력의 기본 형태 : 현재 스타일!
  - iostream 헤더 파일의 포함

```
std::cout << 출력 대상;
```

```
std::cout<<출력 대상1<<출력 대상2<<출력 대상3;
```

```
std::cout<<1<<'a'<<"String"<<std::endl;
```

- 입력의 기본 형태 : 과거 스타일!
  - iostream.h 헤더 파일의 포함

```
cin>>입력 변수;
```

```
cin>>입력 변수1>>입력 변수2>>입력 변수3;
```

```
cin>>val1;
```

- 입력의 기본 형태 : 현재 스타일!
  - iostream 헤더 파일의 포함

```
std::cin>>입력 변수;
```

```
std::cin>>입력 변수1>>입력 변수 2>>입력 변수3;
```

```
std::cin>>val1;
```

예제1) 기본적인 C++ 프로그램

```
/*
   소스: SimpleC++.cpp
 cout과 << 연산자를 이용하여 화면에 출력한다
*/

#include <iostream> // cout과 << 연산자 포함

// C++ 프로그램은 main() 함수에서부터 실행을 시작한다
int main() {
   std::cout << "Hello\n"; // 화면에 Hello를 출력하고 다음 줄로 넘어감
   std::cout << "첫 번째 맛보기입니다.";
   return 0; // main() 함수가 종료하면 프로그램이 종료됨
}
```

```
Hello
첫 번째 맛보기입니다.
```

주석문과 main() 함수

- 주석문
  - 개발자가 자유롭게 붙인 특이 사항의 메모, 프로그램에 대한 설명
  - 프로그램의 실행에 영향을 미치지 않음
    - 여러 줄 주석문 - /* … */
    - 한 줄 주석문 - //를 만나면 이 줄의 끝까지 주석으로 처리

- main() 함수
  - C++ 프로그램의 실행을 시작하는 함수
    - main() 함수가 종료하면 C++ 프로그램 종료
  - main() 함수의 C++ 표준 모양

```
int main() { // main()의 리턴 타입 int
    ……
    return 0; // 0이 아닌 다른 값으로 리턴 가능
}
```

```
void main() { // 표준 아님
    ……
}
```

  - main()에서 return문 생략 가능

```
int main() {
    ……
    // return 0; // 개발자의 편리를 위해 return 문 생략 가능
}
```

#include <iostream>

- #include <iostream>
  - 전처리기(C++ Preprocessor)에게 내리는 지시
    - <iostream> 헤더 파일을 컴파일 전에 소스에 확장하도록 지시
- <iostream> 헤더 파일
  - 표준 입출력을 위한 클래스와 객체, 변수 등이 선언됨
    - ios, istream, ostream, iostream 클래스 선언
    - cout, cin, <<, >> 등 연산자 선언

```
#include <iostream>
....
   std::cout << "Hello\n";
   std::cout << "첫 번째 맛보기입니다.";
```

화면 출력

- cout과 << 연산자 이용

```
std::cout << "Hello\n"; // 화면에 Hello를 출력하고 다음 줄로 넘어감
std::cout << "첫 번째 맛보기입니다.";
```

- cout 객체
  - **스크린** 출력 장치에 연결된 **표준** C++ 출력 스트림 객체
  - <iostream> 헤더 파일에 선언
  - std 이름 공간에 선언: **std::cout**으로 사용

- << 연산자
  - 스트림 삽입 연산자(stream insertion operator)
    - C++ 기본 산술 시프트 연산자(<<)가 스트림 삽입 연산자로 재정의 됨
    - ostream 클래스에 구현됨
    - 오른쪽 피연산자를 왼쪽 스트림 객체에 삽입
    - cout 객체에 연결된 화면에 출력
  - 여러 개의 << 연산자로 여러 값 출력

```
std::cout << "Hello\n" << "첫 번째 맛보기입니다.";
```

예제 2) cout과 <<를 이용한 화면 출력

```
#include <iostream>

double area(int r); // 함수의 원형 선언

double area(int r) { // 함수 구현
   return 3.14*r*r; // 반지름 r의 원면적 리턴
}

int main() {
   int n=3;
   char c='#';
   std::cout << c << 5.5 << '-' << n << "hello" << true << std::endl;
   std::cout << "n + 5 = " << n + 5 << '\n';
   std::cout << "면적은 " << area(n); // 함수 area()의 리턴 값 출력
}
```

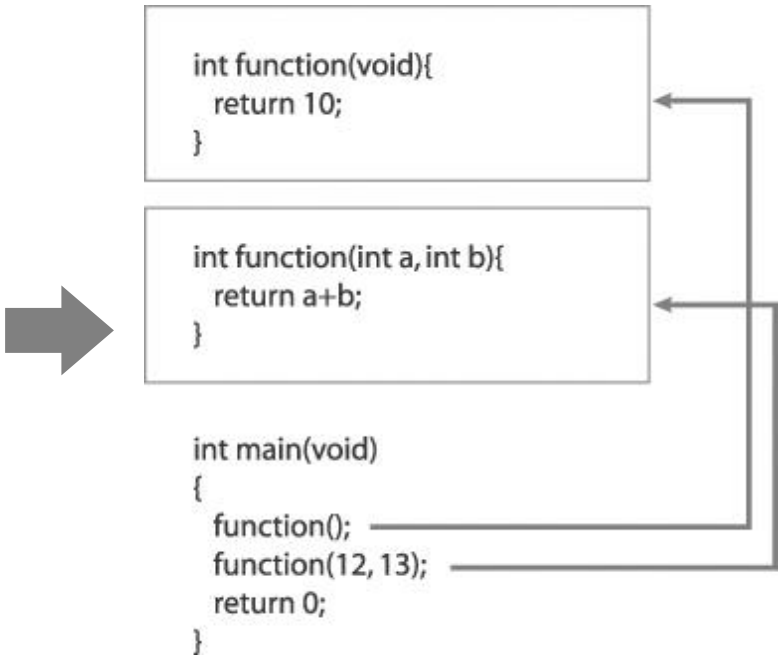
true는 1로 출력됨

```
#5.5-3hello1
n + 5 = 8
면적은 28.26
```

함수 오버로딩

- 함수 오버로딩 (이름의 중복)
  - 파일의 확장자는 .C이다! 무엇이 문제?

```
int function(void){
   return 10;
}

int function(int a, int b){
   return a+b;
}

int main(void)
{
   function();
   function(12, 13);
   return 0;
}
```



함수 오버로딩

- 함수 오버로딩이란?
  - 동일한 이름의 함수를 중복해서 정의하는 것!
- 함수 오버로딩의 조건
  - 매개 변수의 개수 혹은 타입이 일치하지 않는다
- 함수 오버로딩이 가능한 이유
  - 호출할 함수를 매개 변수의 정보까지 참조해서 호출
  - 함수의 이름 + 매개 변수의 정보

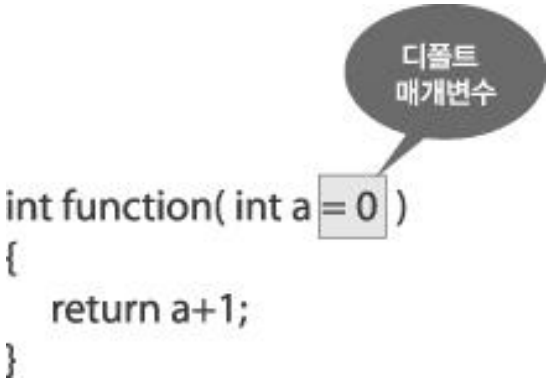
- **함수 오버로딩의 예**

```
int function1(int n)
{…}

int function1(char c)
{…}
```

```
int function2(int v)
{…}

int function2(int v1, int v2)
{…}
```

디폴트 매개 변수

- **디폴트 매개 변수란?**
  - 전달되지 않은 인자를 대신하기 위한 기본 값이 설정되어 있는 변수

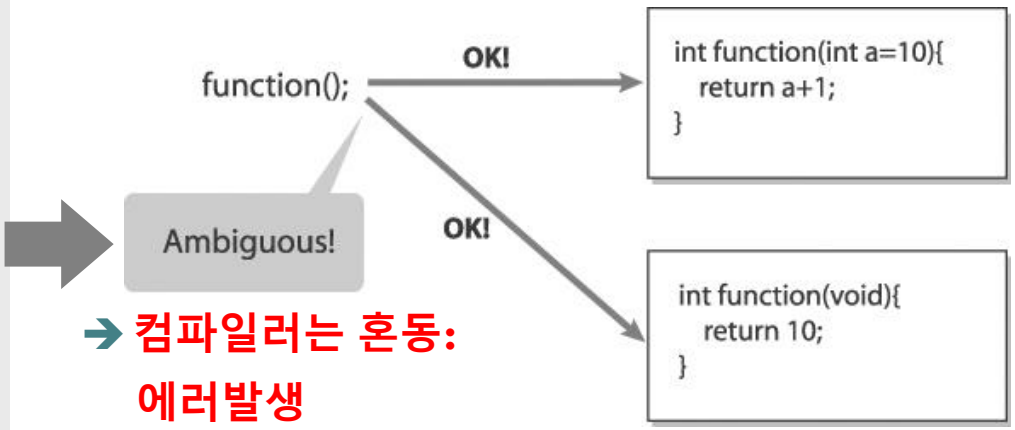


디폴트 매개변수 vs 함수 오버로딩

디폴트 매개변수와 함수오버로딩을 잘못 정의 한 예

```
#include<iostream>

int function(int a=10){
   return a+1;
}

int function(void){
   return 10;
}

int main(void)
{
   std::cout<<function(10)<<std::endl;
   return 0;
}
```

- **main 함수를 변경**



➔ 컴파일러는 혼동: 에러발생

인-라인 함수

- 매크로 함수를 통한 인-라인
  - 인-라인화된 함수
  - 장점! : 실행 속도의 향상
  - 단점! : 구현의 어려움

```
#include <iostream>
#define SQUARE(x) ((x)*(x))

int main(void)
{
   std::cout<< SQUARE(5) <<std::endl;
   return 0;
}
```

- **inline 선언에 의한 함수의 인-라인화**
  - 컴파일러에 의해서 처리
  - 매크로 함수의 장점을 그대로 반영
  - 구현의 용이성 제공
  - 컴파일러에게 최적화의 기회 제공

```
#include <iostream>
inline int SQUARE(int x)
{
   return x*x;
}

int main(void)
{
   std::cout<<SQUARE(5)<<std::endl;
   return 0;
}
```

이름공간(namespace)

- 이름 공간이란?
  - 공간에 이름을 주는 행위!
  - "202호에 사는 철수야"

```
#include <iostream>

void function(void)
{
   std::cout<<"A.com에서 정의한 함수"<<std::endl;
}

void function(void)
{
   std::cout<<"B.com에서 정의한 함수"<<std::endl;
}

int main(void)
{
   function();
   return 0;
}
```

namespace (이름공간)

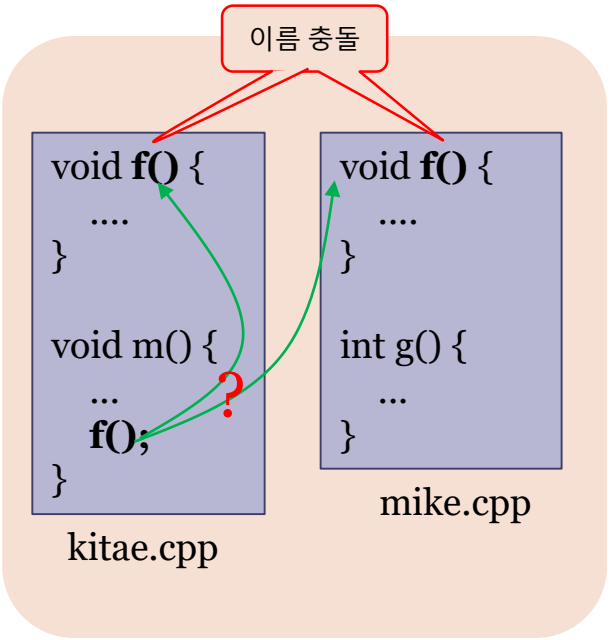
- 이름(identifier) 충돌이 발생하는 경우
  - 여러 명이 서로 나누어 프로젝트를 개발하는 경우
  - 오픈 소스 혹은 다른 사람이 작성한 소스나 목적 파일을 가져와서 컴파일 하거나 링크하는 경우
  - 해결하는데 많은 시간과 노력이 필요

- namespace 키워드
  - 이름 충돌 해결
    - 2003년 새로운 C++ 표준에서 도입
  - 개발자가 자신만의 이름 공간을 생성할 수 있도록 함
    - 이름 공간 안에 선언된 이름은 다른 이름공간과 별도 구분

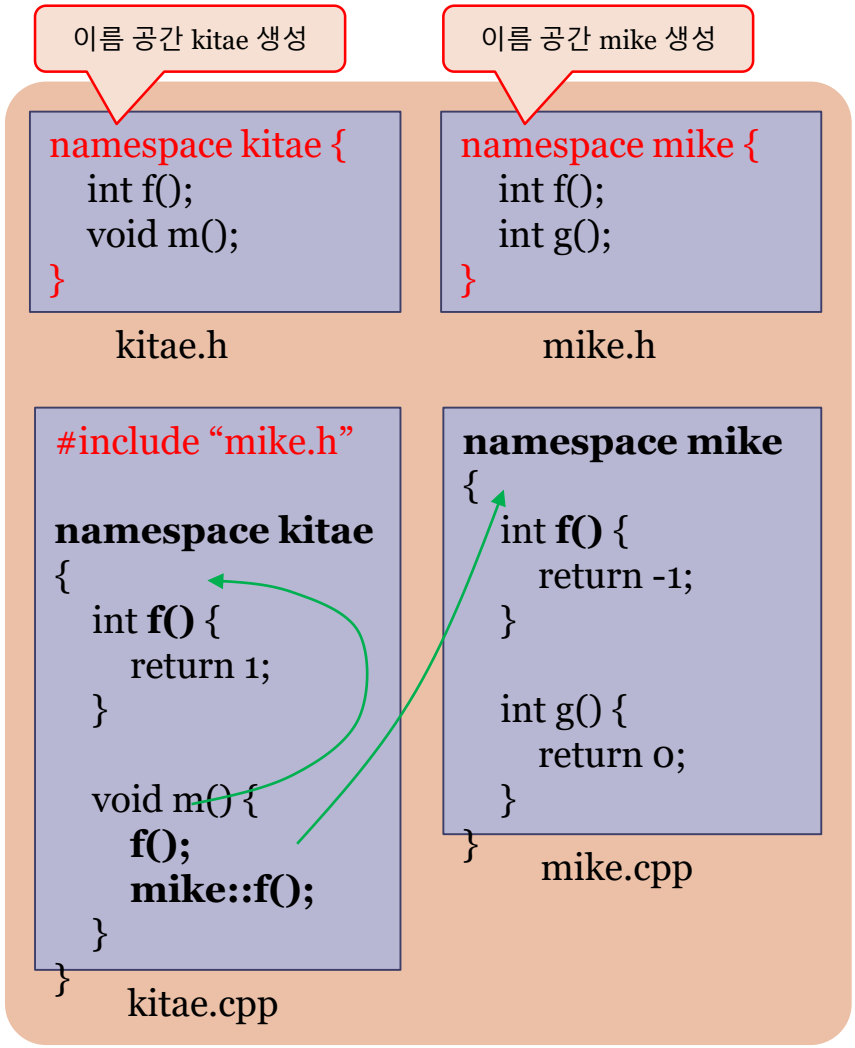
- 이름 공간 생성 및 사용

```
namespace kitae { // kitae 라는 이름 공간 생성
   ..... // 이 곳에 선언된 모든 이름은 kitae 이름 공간에 생성된 이름
}
```

  - 이름 공간 사용
    - 이름 공간 :: 이름

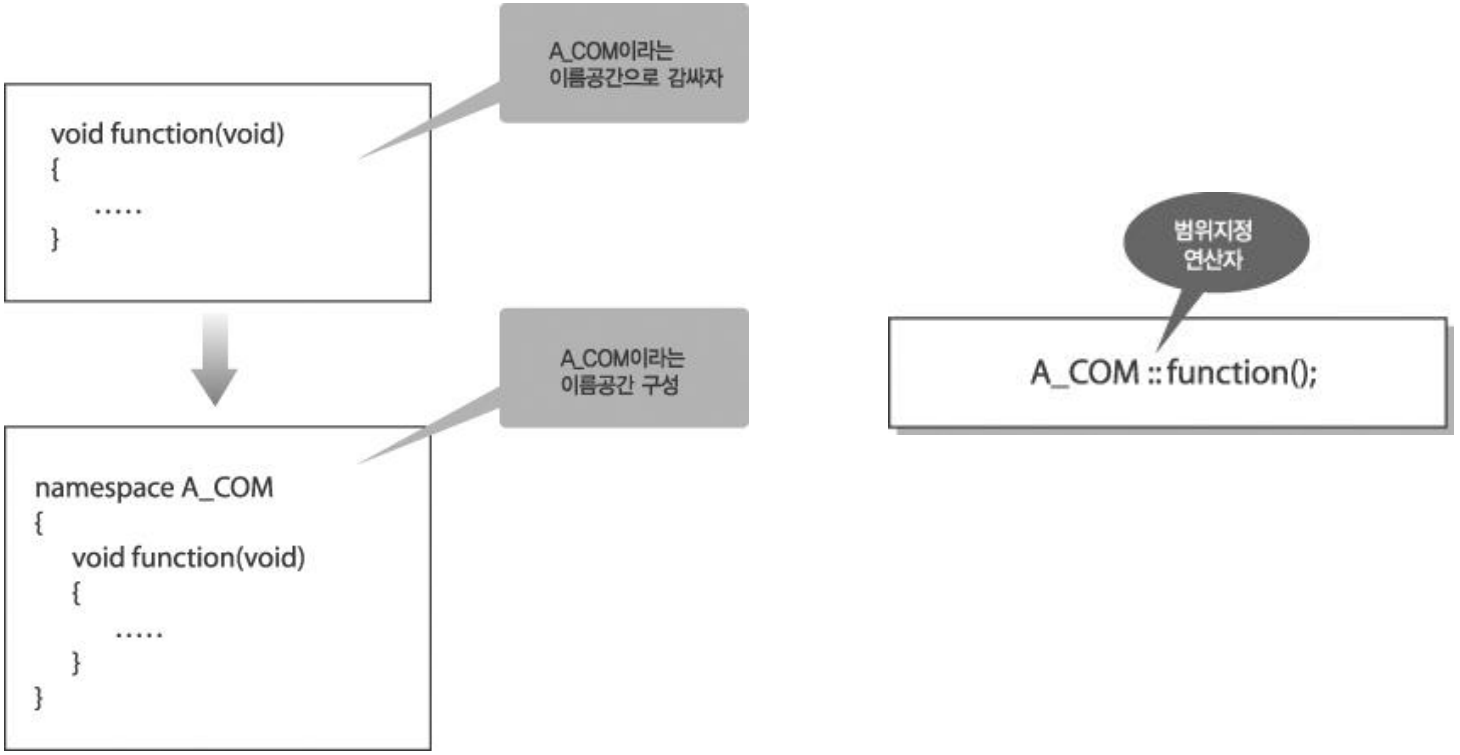


(a) kitae와 mike에 의해 작성된 소스를 합치면 f() 함수의 이름 충돌. 컴파일 오류 발생



(b) 이름 공간을 사용하여 f() 함수 이름의 충돌 문제 해결

- 이름 공간의 적용


A_COM이라는
이름공간으로 감싸자
void function(void)
{
.....
}
A_COM이라는
이름공간 구성
namespace A_COM
{
void function(void)
{
.....
}
}
범위지정
연산자
A_COM :: function();

- 아하! std란 namespace!

- **편의를 위한 using 선언!**

```
using A_COM::function;
```

```
using namespace A_COM;
```

예제) C++ 프로그램에서 키 입력 받기

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

int main() {
   cout << "너비를 입력하세요>>";

   int width;
   cin >> width; // 키보드로부터 너비를 읽어 width 변수에 저장

   cout << "높이를 입력하세요>>";

   int height;
   cin >> height; // 키보드로부터 높이를 읽어 height 변수에 저장

   int area = width*height;   // 사각형의 면적 계산
   cout << "면적은 " << area << "\n"; // 면적을 출력하고 다음 줄로 넘어감
}
```

```
너비를 입력하세요>>3
높이를 입력하세요>>5
면적은 15
```

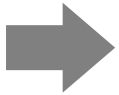
표준 C++ 헤더 파일은 확장자가 없다

- 표준 C++에서 헤더 파일 확장자 없고, std 이름 공간 적시
  - #include <iostream>
  - using namespace std;

언어	헤더 파일 확장자	사례	설명
C	.h	〈string.h〉	C/C++ 프로그램에서 사용 가능
C++	확장자 없음	〈cstring〉	using namespace std;와 함께 사용해야 함

- 범위 지정 연산자 기반 전역 변수 접근

```
int val=100;

int main(void)
{
   int val=100;
   val+=1;
   return 0;
}
```



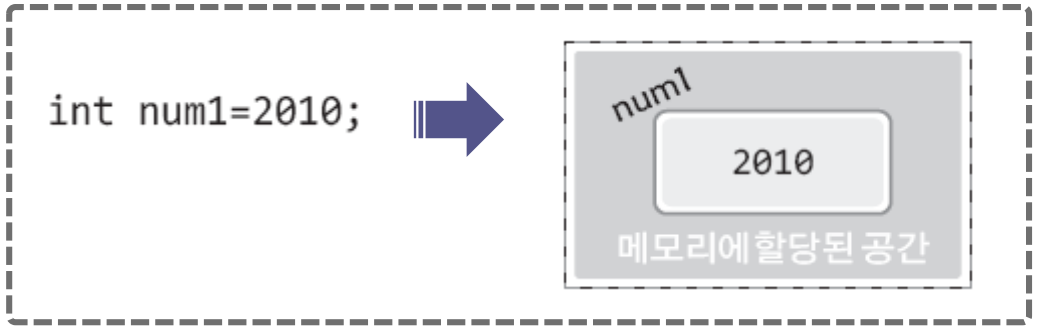
```
int val=100;

int main(void)
{
   int val=100;
   ::val+=1;
   return 0;
}
```

실행 결과 >

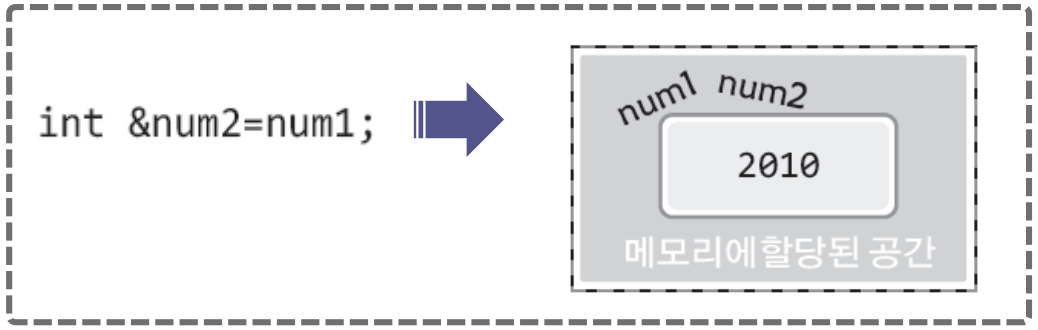
전역변수	지역변수
100	101

전역변수	지역변수
101	100

참조자(Reference)의 이해



변수의 선언으로 인해서 num1이라는 이름으로 메모리 공간이 할당된다.



참조자의 선언으로 인해서 num1의 메모리 공간에 num2라는 이름이 추가로 붙게 된다.

참조자는 기존에 선언된 변수에 붙이는 '별칭'이다. 그리고 이렇게 참조자가 만들어지면 이는 변수의 이름과 사실상 차이가 없다.

예제

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

int main()
{
        int var;
        int &ref = var;                    // 참조자선언

        var = 10;
        cout << "var의값: " << var << endl;
        cout << "ref의값: " << ref << endl;

        ref = 20;                  // ref의 값을 변경하면 var의 값도 변경된다
        cout << "var의값: " << var << endl;
        cout << "ref의값: " << ref << endl;

        return 0;
}
```

```
var의 값: 10
ref의 값: 10
var의 값: 20
ref의 값: 20
```

참조자 관련 예제와 참조자의 선언

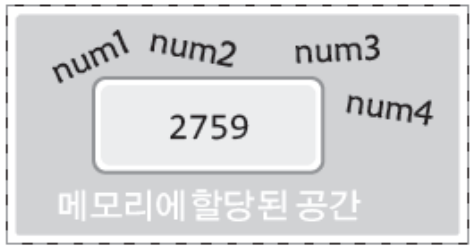
```
int main(void)
{
    int num1=1020;
    int &num2=num1;
    num2=3047;
    cout<<"VAL: "<<num1<<endl;
    cout<<"REF: "<<num2<<endl;
    cout<<"VAL: "<<&num1<<endl;
    cout<<"REF: "<<&num2<<endl;
    return 0;
}
```

num2는 num1의 참조자이다. 따라서 이후부터는 num1으로 하는 모든 연산은 num2로 하는것과 동일한 결과를 보인다

실행결과

```
VAL: 3047
REF: 3047
VAL: 0012FF60
REF: 0012FF60
```

```
int num1=2759;
int &num2=num1;
int &num3=num2;
int &num4=num3;
```



참조자의 수에는 제한이 없으며, 참조자를 대상으로 참조자를 선언하는 것도 가능하다

참조자의 선언 가능 범위

```
int &ref=20;        (×)
```

상수 대상으로의 참조자 선언은 불가능하다.

```
int &ref;           (×)
```

참조자는 생성과 동시에 누군가를 참조해야 한다.

```
int &ref=NULL;      (×)
```

포인터처럼 NULL로 초기화하는 것도 불가능하다.

불가능한 참조자의 선언의 예

정리하면, 참조자는 선언과 동시에 누군가를 참조해야 하는데, 그 참조의 대상은 기본적으로 변수가 되어야 한다. 그리고 참조자는 참조의 대상을 변경할 수 없다

```
int main(void)
{
    int arr[3]={1, 3, 5};
    int &ref1=arr[0];
    int &ref2=arr[1];
    int &ref3=arr[2];

    cout<<ref1<<endl;
    cout<<ref2<<endl;
    cout<<ref3<<endl;
    return 0;
}
```

변수의 성향을 지니는 대상이라면 참조자의 선언이 가능하다

배열의 요소 역시 변수의 성향을 지니기 때문에 참조자의 선언이 가능하다

```
1
3
5
```

실행결과

포인터 변수 대상의 참조자 선언

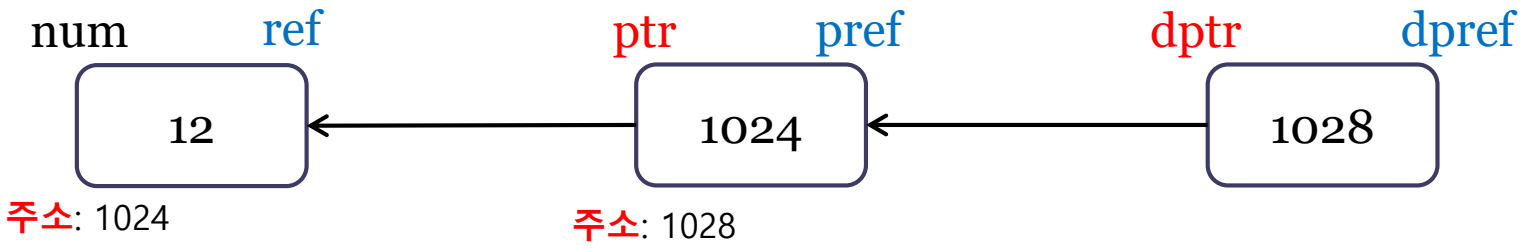
```
int main(void)
{
    int num=12;
    int *ptr=&num;
    int **dptr=&ptr;

    int &ref=num;
    int *(&pref)=ptr;
    int **(&dpref)=dptr;

    cout<<ref<<endl;
    cout<<*pref<<endl;
    cout<<**dpref<<endl;
    return 0;
}
```

ptr과 dptr 역시 변수이다. 다만 주소 값을 저장하는 포인터 변수일 뿐이다. 따라서 이렇듯 참조자의 선언이 가능하다

실행결과

```
12
12
12
```

참조자와 포인터의 활용



- 포인터 변수: ptr, dptr
- 참조자: ref, pref, dpref

참조자와 포인터 비교

- **참조자**는 반드시 선언과 동시에 초기화

```
int &ref;              // 오류!
```

- **포인터**는 변경될 수 있지만 **참조자**는 변경이 불가능하다

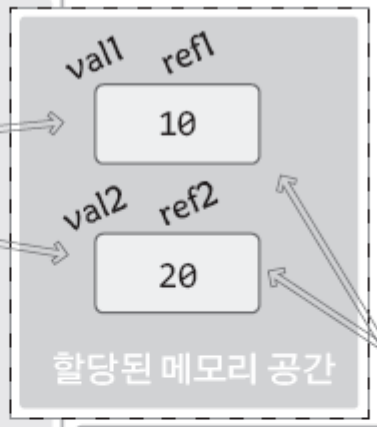
```
int &ref = var1;
ref = var2;            // 오류!
```

- **참조자**를 상수로 초기화하면 컴파일

```
int &ref = 10;         // 오류!
```

참조자를 이용한 Call-by-reference

```
int main(void)
{
  int val1=10;
  int val2=20;
  SwapByRef2(val1, val2);
  cout<<"val1: "<<val1<<endl;
  cout<<"val2: "<<val2<<endl;
  return 0;
}
```




```
void SwapByRef2(int &ref1, int &ref2)
{
  int temp=ref1;
  ref1=ref2;
  ref2=temp;
}   // Call-by-reference
```

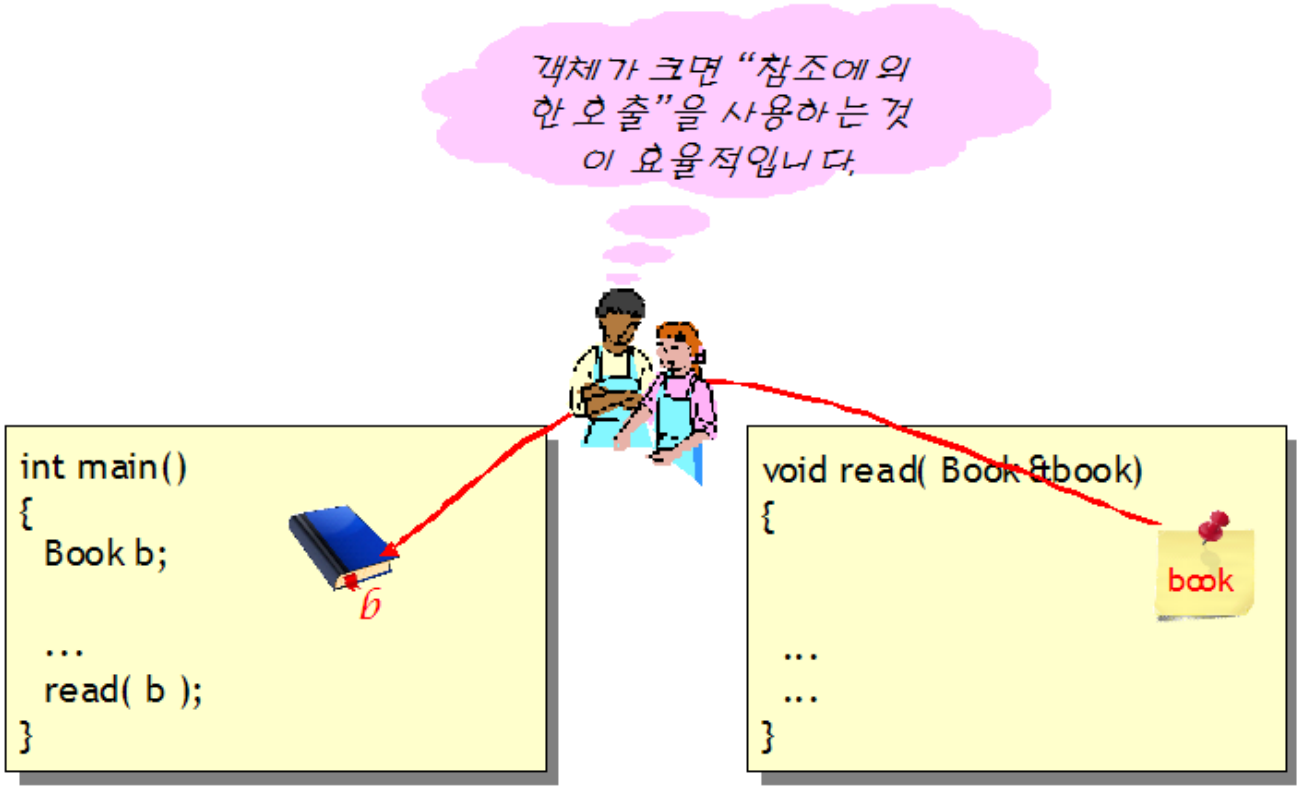
매개변수는 함수가 호출될 때 선언이 되는 변수이므로, 함수호출의 과정에서 선언과 동시에 전달되는 대상으로 초기화된다

즉, 매개변수에 선언된 참조자는 여전히 선언과 동시에 초기화된다

참조자 기반의 Call-by-reference!

참조자를 통한 효율성 향상

- 객체의 크기가 큰 경우, 복사는 시간이 많이 걸린다. 이때는 참조자로 처리하는 것이 유리



const 참조자

함수의 호출 형태

```
int num=24;
HappyFunc(num);
```

함수의 정의 형태

```
void HappyFunc(int &ref) { . . . . }
```

함수의 정의형태와 함수의 호출형태를 보아도 값의 변경유무를 알 수 없다! 이를 알려면 HappyFunc 함수의 몸체 부분을 확인해야 한다. 그리고 이는 큰 단점이다!

```
void HappyFunc(const int &ref) { . . . . }
```

함수 HappyFunc 내에서 참조자 ref를 이용한 값의 변경은 허용하지 않겠다! 라는 의미!

함수 내에서 참조자를 통한 값의 변경을 진행하지 않을 경우 참조자를 const로 선언해서, 다음 두 가지 장점을 얻도록 하자!

1. 함수의 원형 선언만 봐도 값의 변경이 일어나지 않음을 판단할 수 있다
2. 실수로 인한 값의 변경이 일어나지 않는다

동적 메모리 할당: new & delete

• int형 변수의 할당	`int * ptr1=new int;`
• double형 변수의 할당	`double * ptr2=new double;`
• 길이가 3인 int형 배열의 할당	`int * arr1=new int[3];`
• 길이가 7인 double형 배열의 할당	`double * arr2=new double[7];`

malloc을 대한하는 메모리의 동적 할당방법!

크기를 바이트 단위로 계산하는 일을 거치지 않아도 된다!

• 앞서 할당한 int형 변수의 소멸	`delete ptr1;`
• 앞서 할당한 double형 변수의 소멸	`delete ptr2;`
• 앞서 할당한 int형 배열의 소멸	`delete []arr1;`
• 앞서 할당한 double형 배열의 소멸	`delete []arr2;`

free를 대신하는 메모리의 해제방법!

new 연산자로 할당된 메모리 공간은 반드시 delete 함수호출을 통해서 소멸해야 한다! 특히 이후에 공부하는 객체의 생성 및 소멸 과정에서 호출하게 되는 new & delete 연산자의 연산자의 연산특성은 malloc & free와 큰 차이가 있다!

C++의 표준헤더: c를 더하고 .h를 빼라

```
#include <stdio.h>     →  #include <cstdio>
#include <stdlib.h>    →  #include <cstdlib>
#include <math.h>      →  #include <cmath>
#include <string.h>    →  #include <cstring>
```

이렇듯 C언어에 대응하는 C++ 헤더파일 이름의 정의에는 일정한 규칙이 적용되어 있다

```
int abs(int num);
```

표준 C의 abs 함수 ⟷

```
long abs(long num);
float abs(float num);
double abs(double num);
long double abs(long double num);
```

대응하는 C++의 표준 abs 함수

이렇듯, 표준 C에 대응하는 표준 C++ 함수는 C++ 문법을 기반으로 변경 및 확장되었다. 따라서 가급적이면 C++의 헤더파일을 포함하여, C++의 표준함수를 호출해야 한다

실전문제1) 최대한의 사탕 사기

- 철수가 가지고 있는 돈으로 최대한의 사탕을 사려고 한다. 현재 1000원이 있고 사탕의 가격이 300원이라고 하자. 최대한 살 수 있는 사탕의 개수와 나머지 돈은 얼마인가?

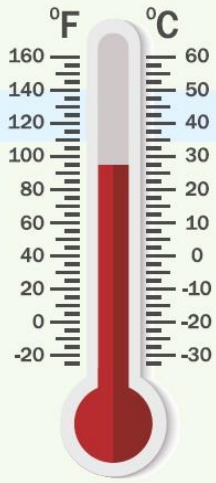
정답)

```
#include <iostream>
using namespace std;
int main()
{
		int money;
		int candy_price;

		cout << "현재 가지고 있는 돈: ";
		cin >> money;
		cout << "캔디의 가격: ";
		cin >> candy_price;

		// 최대한 살 수 있는 사탕 수
		int n_candies = money / candy_price;
		cout << "최대로 살 수 있는 캔디의 개수=" << n_candies << endl;

		// 사탕을 구입하고 남은 돈
		int change = money % candy_price;
		cout << "캔디 구입 후 남은 돈=" << change << endl;
		return 0;
}
```

실전문제2) 화씨 섭씨 변환 프로그램

- 우리나라는 섭씨 온도를 사용하지만 미국에서는 화씨 온도를 사용한다. 화씨 온도를 섭씨 온도로 바꾸는 프로그램을 작성하여 보자

```
c_temp = (5.0 / 9.0) * (f_temp - 32);
```




```
C:₩Windows₩system32₩cmd.exe
화씨온도 60도는 섭씨온도 15.5556입니다.
계속하려면 아무 키나 누르십시오 . . .
```

정답)

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

int main()
{
	double f_temp = 60;
	double c_temp;

	c_temp = (5.0 / 9.0) * (f_temp - 32);
	cout << "화씨온도 " << f_temp << "도는 섭씨온도 " << c_temp << "입니다." << endl;
		return 0;
}
```

참고문헌

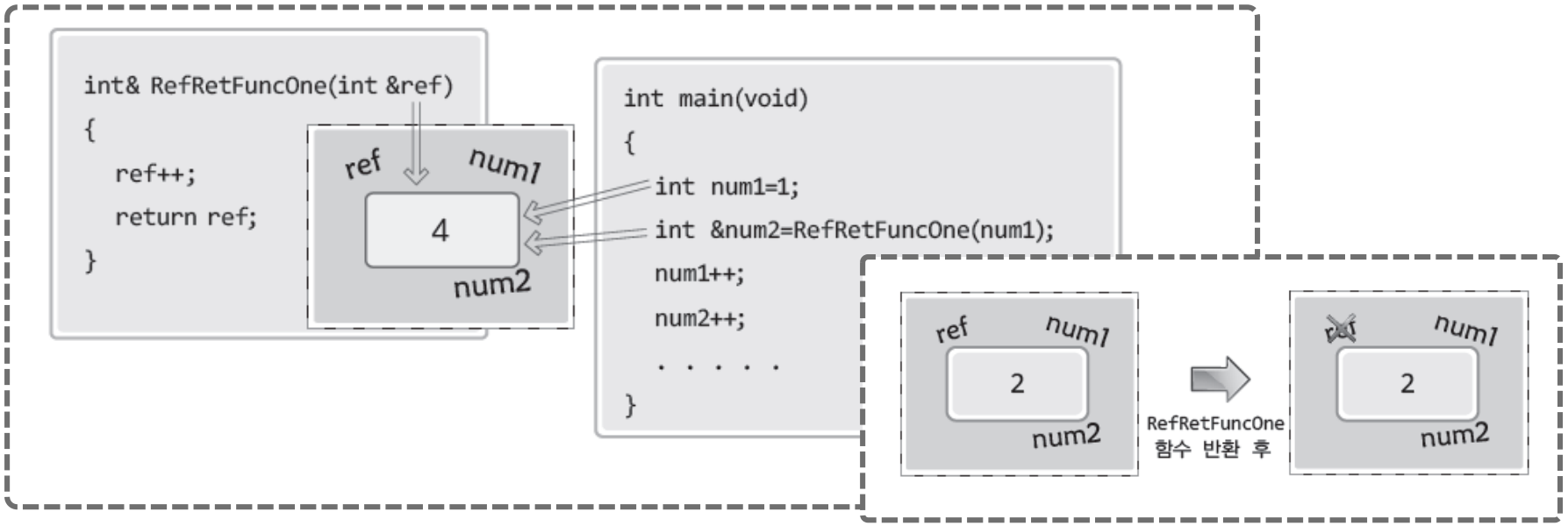
- 뇌를 자극하는 C++ 프로그래밍, 이현창, 한빛미디어, 2011
- 열혈 C++ 프로그래밍(개정판), 윤성우, 오렌지미디어, 2012
- 객체지향 원리로 이해하는 ABSOLUTE C++ , 최영근 외 4명 , 교보문고, 2013
- C++ ESPRESSO, 천인국, 인피니티북스, 2011
- 명품 C++ Programming, 황기태 , 생능출판사, 2018
- 어서와 C++는 처음이지, 천인국, 인피니티북스, 2018

Q&A

추가 참고자료

-참조자 부연 설명
-포인터사용 없이 힙에 접근 하기

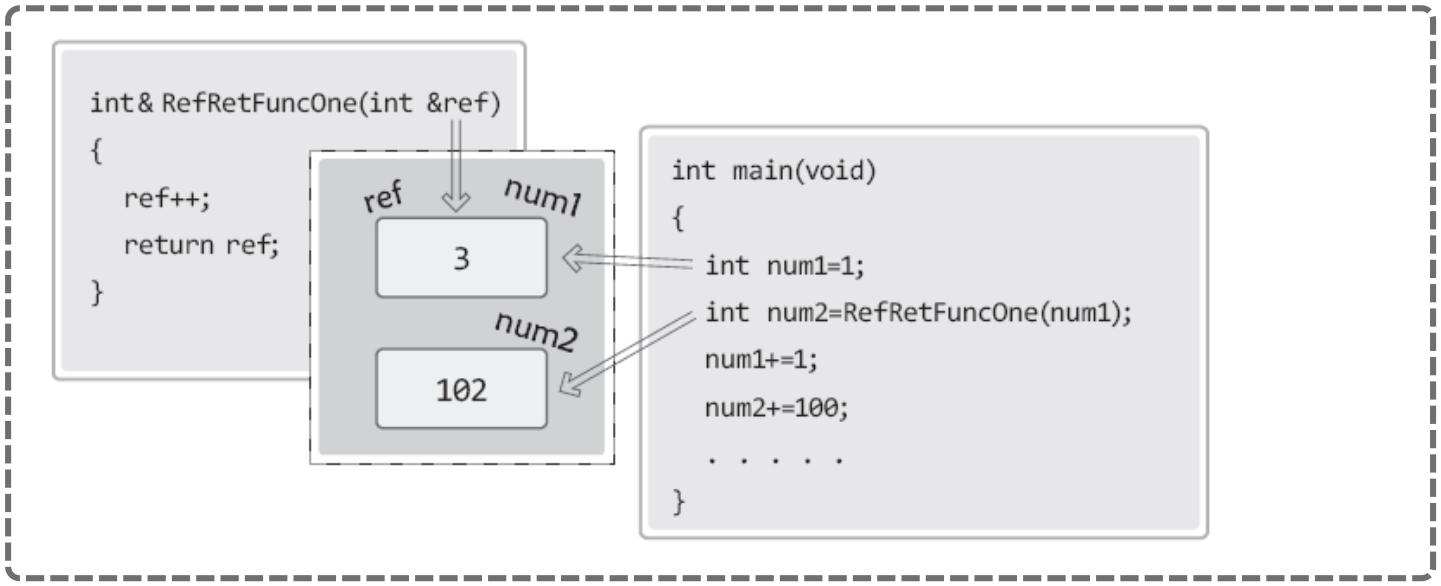
반환형이 참조이고 반환도 참조로 받는 경우



반환의 과정에서 일어나는 일은 다음의 경우와 같다.

```
int num1=1;
int &ref=num1;      // 인자의 전달과정에서 일어난 일
int &num2=ref;      // 함수의 반환과 반환 값의 저장에서 일어난 일
```

반환형은 참조이나 반환은 변수로 받는 경우



반환의 과정에서 일어나는 일은 다음의 경우와 같다.

```
int num1=1;
int &ref=num1;      // 인자의 전달과정에서 일어난 일
int num2=ref;       // 함수의 반환과 반환 값의 저장에서 일어난 일
```

참조를 대상으로 값을 반환하는 경우

```
int RefRetFuncTwo(int &ref)
{
    ref++;
    return ref;
}
```

```
int main(void)
{
    int num1=1;
    int num2=RefRetFuncTwo(num1);
    num1+=1;
    num2+=100;
    cout<<"num1: "<<num1<<endl;
    cout<<"num2: "<<num2<<endl;
    return 0;
}
```

참조자를 반환하건, 변수에 저장된 값을 반환하건, 반환형이 참조형이 아니라면 차이는 없다! 어차피 참조자가 참조하는 값이나 변수에 저장된 값이 반환되므로!

- `int num2=RefRetFuncOne(num1);` (○)
- `int &num2=RefRetFuncOne(num1);` (○)

반환형이 참조형인 경우에는 반환되는 대상을 참조자로 그리고 변수로 받을 수 있다.

- `int num2=RefRetFuncTwo(num1);` (○)
- `int &num2=RefRetFuncTwo(num1);` (×)

그러나 반환형이 값의 형태라면, 참조자로 그 값을 받을 수 없다!

잘못된 참조의 반환

```
int& RetuRefFunc(int n)
{
    int num=20;
    num+=n;
    return num;
}
```

이와 같이 지역변수를 참조의 형태로 반환하는 것은 문제의 소지가 된다. 따라서 이러한 형태로는 함수를 정의하면 안 된다

에러의 원인! ref가 참조하는 대상이 소멸된다!

```
int &ref=RetuRefFunc(10);
```

const 참조자의 또 다른 특징

```
const int num=20;
int &ref=num;
ref+=10;
cout<<num<<endl;
```

에러의 원인! 이를 허용한다는 것은 ref 를 통한 값의 변경을 허용한다는 뜻이 되고, 이는 num을 const로 선언하는 이유를 잃게 만드는 결과이므로!

해결책!

```
const int num=20;
const int &ref=num;
const int &ref=50;
```

따라서 한번 const 선언이 들어가기 시작하면 관련해서 몇몇 변수들이 const 로 선언되어야 하는데, 이는 프로그램의 안전성을 높이는 결과로 이어지기 때문에, const 선언을 빈번히 하는 것은 좋은 습관이라 할 수 있다

어떻게 참조자가 상수를 참조하냐고요!

const 참조자는 상수를 참조할 수 있다.

이유는,

이렇듯, 상수를 const 참조자로 참조할 경우, 상수를 메모리 공간에 임시적으로 저장하기 때문이다! 즉, 행을 바꿔도 소멸시키지 않는다

이러한 것이 가능하도록 한 이유!

```
int Adder(const int &num1, const int &num2)
{
    return num1+num2;
}
```

이렇듯 매개변수 형이 참조자인 경우에 상수를 전달할 수 있도록 하기 위함이 바로 이유이다!

포인터를 사용하지 않고 힙에 접근하기

```
int *ptr=new int;
int &ref=*ptr;          // 힙 영역에 할당된 변수에 대한 참조자 선언
ref=20;
cout<<*ptr<<endl;       // 출력결과는 20!
```

변수의 성향을 지니는(값의 변경이 가능한) 대상에 대해서는
참조자의 선언이 가능하다

C언어의 경우 힙 영역으로의 접근을 위해서는 반드시 포인터를 사용해야만 했다. 하지만 C++에서는 참조자를 이용한 접근도 가능하다!

Q&A