Chapter 01. C++ 기초

(C언어복습 및 C++ 기초 문법)

박 종 혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

C언어 복습

구조체, 포인터, 기타

C언어의 복습을 유도하는 확인학습 문제1

[문제 1] 키워드 const의 의미

키워드 const는 어떠한 의미를 갖는가? 다음 문장들을 대상으로 이를 설명해보자.

- `const int num=10;`
- `const int * ptr1=&val1;`
- `int * const ptr2=&val2;`
- `const int * const ptr3=&val3;`

C언어의 복습을 유도하는 확인학습 문제1

[문제 1] 키워드 const의 의미

키워드 const는 어떠한 의미를 갖는가? 다음 문장들을 대상으로 이를 설명해보자.

- `const int num=10;`
- `const int * ptr1=&val1;`
- `int * const ptr2=&val2;`
- `const int * const ptr3=&val3;`

문제 1의 답안

- const int num=10;
  - ➡ 변수 num을 상수화!
- const int * ptr1=&val1;
  - ➡ 포인터 ptr1을 이용해서 val1의 값을 변경할 수 없음
- int * const ptr2=&val2;
  - ➡ 포인터 ptr2가 상수화 됨
- const int * const ptr3=&val3;
  - ➡ 포인터 ptr3가 상수화 되었으며, ptr3를 이용해서 val3의 값을 변경할 수 없음

C언어의 복습을 유도하는 확인학습 문제2

[문제 2] 실행중인 프로그램의 메모리 공간

실행중인 프로그램은 운영체제로부터 메모리 공간을 할당 받는데, 이는 크게 데이터, 스택, 힙 영역으로 나뉜다. 각각의 영역에는 어떠한 형태의 변수가 할당되는지 설명해보자. 특히 C언어의 malloc과 free 함수와 관련해서도 설명해보자.

C언어의 복습을 유도하는 확인학습 문제2

[문제 2] 실행중인 프로그램의 메모리 공간

실행중인 프로그램은 운영체제로부터 메모리 공간을 할당 받는데, 이는 크게 데이터, 스택, 힙 영역으로 나뉜다. 각각의 영역에는 어떠한 형태의 변수가 할당되는지 설명해보자. 특히 C언어의 malloc과 free 함수와 관련해서도 설명해보자.

문제 2의 답안

• 데이터	전역변수가 저장되는 영역
• 스택	지역변수 및 매개변수가 저장되는 영역
• 힙	malloc 함수호출에 의해 프로그램이 실행되는 과정에서 동적으로 할당이 이뤄지는 영역
• malloc & free	malloc 함수호출에 의해 할당된 메모리 공간은 free 함수호출을 통해서 소멸하지 않으면 해제되지 않는다.

C언어의 복습을 유도하는 확인학습 문제3

[문제 3] Call-by-value vs. Call-by-reference

함수의 호출형태는 크게 '값에 의한 호출(Call-by-value)'과 '참조에 의한 호출(Call-by-reference)'로 나뉜다. 이 둘을 나누는 기준이 무엇인지, 두 int형 변수의 값을 교환하는 Swap 함수를 예로 들어가면서 설명해보자.

C언어의 복습을 유도하는 확인학습 문제3

[문제 3] Call-by-value vs. Call-by-reference

함수의 호출형태는 크게 '값에 의한 호출(Call-by-value)'과 '참조에 의한 호출(Call-by-reference)'로 나뉜다. 이 둘을 나누는 기준이 무엇인지, 두 int형 변수의 값을 교환하는 Swap 함수를 예로 들어가면서 설명해보자.

문제 3의 답안

```
void SwapByValue(int num1, int num2)
{
    int temp=num1;
    num1=num2;
    num2=temp;
}   // Call-by-value

void SwapByRef(int * ptr1, int * ptr2)
{
    int temp=*ptr1;
    *ptr1=*ptr2;
    *ptr2=temp;
}   // Call-by-reference
```

Call-by-value & Call-by-reference

```
void SwapByValue(int num1, int num2)
{
     int temp=num1;
     num1=num2;
     num2=temp;
}    // Call-by-value
```

값을 전달하면서 호출하게 되는 함수이므로 이 함수는 Call-by-value이다. 이 경우 함수 외에 선언된 변수에는 접근이 불가능하다.

```
void SwapByRef(int * ptr1, int * ptr2)
{
     int temp=*ptr1;
     *ptr1=*ptr2;
     *ptr2=temp;
}    // Call-by-reference
```

값은 값이되, 주소 값을 전달하면서 호출하게 되는 함수이므로 이 함수는 Call-by-reference 이다. 이 경우 인자로 전달된 주소의 메모리 공간에 접근이 가능하다!

C언어 학습 시 공부한 내용에 대한 복습이다.

Call-by-address? Call-by-reference!

```
int * SimpleFunc(int * ptr)
{
    return ptr+1;
}
```

포인터 ptr에 전달된 주소 값의 관점에서 보면 이는 Call-by-value이다.

```
int * SimpleFunc(int * ptr)
{
    if(ptr==NULL)
        return NULL;
    *ptr=20;
    return ptr;
}
```

주소 값을 전달 받아서 외부에 있는 메모리 공간에 접근을 했으니 이는 Call-by-reference이다.

C++에는 두 가지 형태의 Call-by-reference가 존재한다. 하나는 주소 값을 이용하는 형태이며, 다른 하나는 참조자를 이용하는 형태이다.

구조체

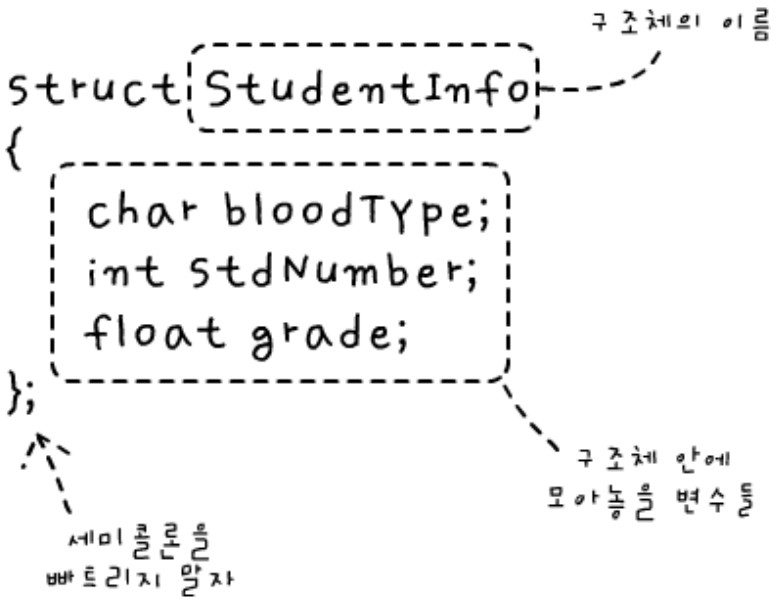
01구조체정의
02구조체변수 정의
03구조체 초기화

구조체의 정의

- 학생의 정보를 보관하기 위한 구조체 정의

```
// 'StudentInfo'라는 이름의 구조체를 정의
struct StudentInfo
{
        char   bloodType;   // 혈액형
        int    stdNumber;   // 학번
        float  grade;       // 평점
};
```

- 구조체를 정의하는 방법

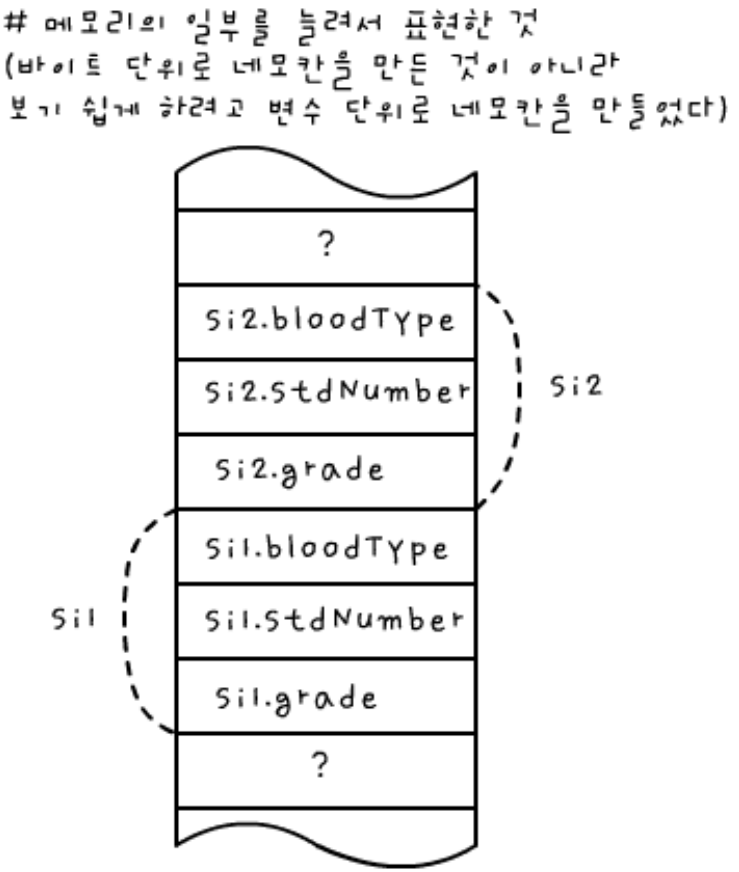


구조체 변수의 정의

```
// StdudentInfo 구조체 타입의 변수 2개를 정의한다.
StudentInfo si1;
StudentInfo si2;
```

- 두 명의 학생 정보를 보관하기 위해서 두 개의 구조체 변수를 정의

- 메모리 구조



구조체 vs 구조체 변수

- 구조체는 구조체 변수의 모양(layout)을 명시하는 설계도와 같은 역할
  - ex) 앞에서 본 구조체 정의의 의미
    - "컴퓨터야, 앞으로 StudentInfo 라는 구조체의 모양(layout)에 대해서 알려줄게. 그 구주체는 bloodType, stdNumber, grade라는 변수가 들어갈거야"

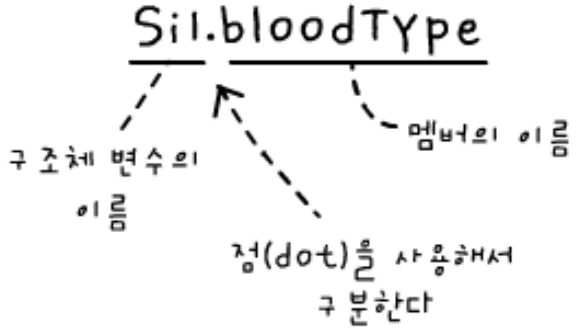
- 구조체 변수를 정의할 때 실제로 정보를 보관할 수 있는 공간이 마련
  - ex) 앞에서 본 구조체 변수 정의의 의미
    - "컴퓨터야, 앞에서 정의한 StudentInfo 라는 구조체 알지? 그 때 알려준 모양(layout)으로 변수를 하나 만들어다오"

멤버에 접근하기

- 구조체에 포함되는 각 변수 - 멤버 혹은 멤버변수
  ex) StudentInfo 구조체의 멤버 변수들
  - bloodType, stdNumber, grade

```
StudentInfo   si1;

si1.bloodType = 'O';
si1.stdNumber = 20031128;
si1.grade = 3.5f;
```

- 멤버 변수의 값을 얻어오거나 변경하는 방법



구조체의 초기화

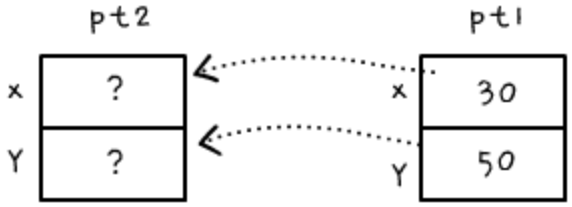
- 구조체 변수를 정의함과 동시에 모든 멤버들을 초기화할 수 있음

```
struct StudentInfo
{
        char bloodType;   // 혈액형
        int stdNumber;    // 학번
        float grade;      // 평점
};

int main()
{
        // StudentInfo 구조체 타입의 변수 2개를 정의한다.
        StudentInfo     si1 = { 'O', 20031128, 3.5f };
        StudentInfo     si2 = { 'A', 19961219, 2.3f };
        .
        .
```

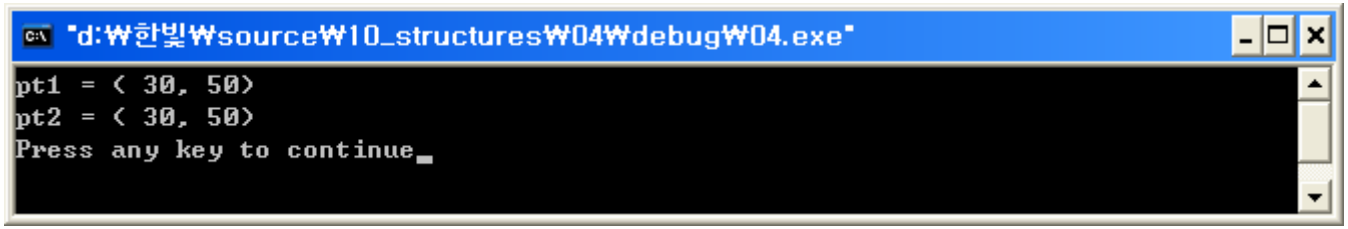
구조체의 대입

- 구조체 변수끼리 대입을 하면 각 멤버 변수들이 1:1로 복사된다.

```
struct Point
{
        int x;      // x 좌표
        int y;      // y 좌표
};

Point pt1 = { 30, 50};
Point pt2;

pt2 = pt1; // pt1을 pt2에 대입한다.

cout << "pt1 = ( " << pt1.x << ", " << pt1.y << ")\n";
cout << "pt2 = ( " << pt2.x << ", " << pt2.y << ")\n";
```



- 실행 결과



구조체 정의 2

- 구조체를 정의하는 동시에 구조체 변수를 정의할 수 있다.

```
struct Point
{
        int x;
        int y;
} pt1 = { 30, 50}, pt2;
```

- 이런 경우에는 구조체의 이름을 생략할 수 있다.

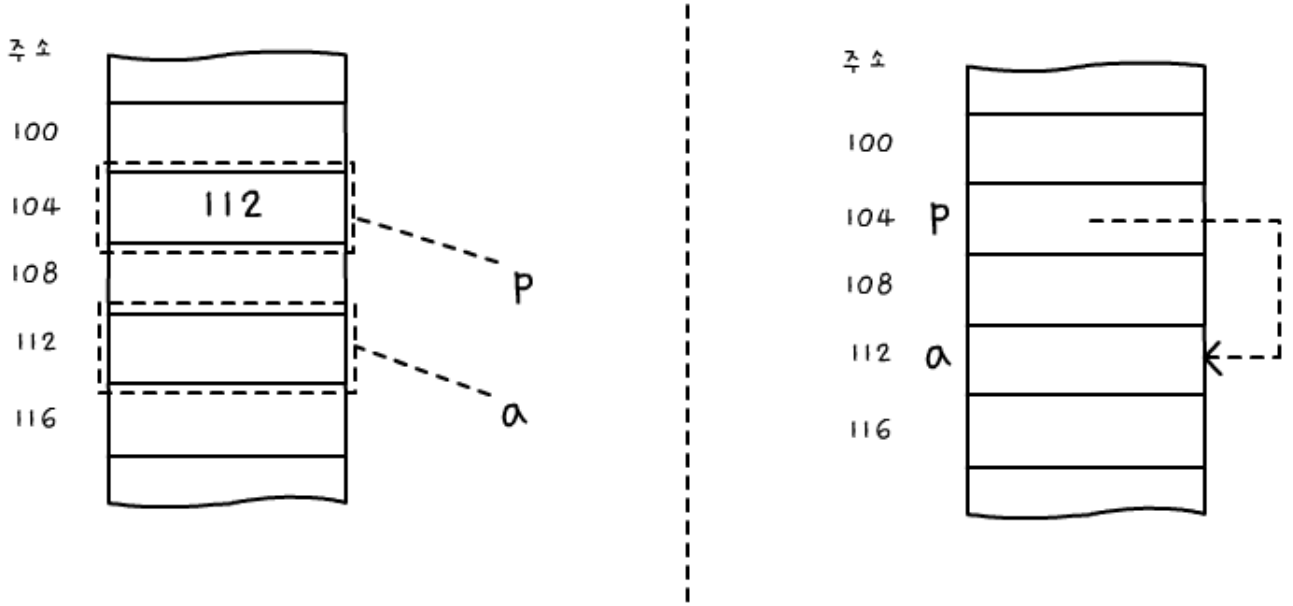
```
struct      // 구조체의 이름 생략
{
        int x;
        int y;
} pt1 = { 30, 50}, pt2;
```

포인터

01 포인터의 기본
02 포인터와 Const

포인터의 개념

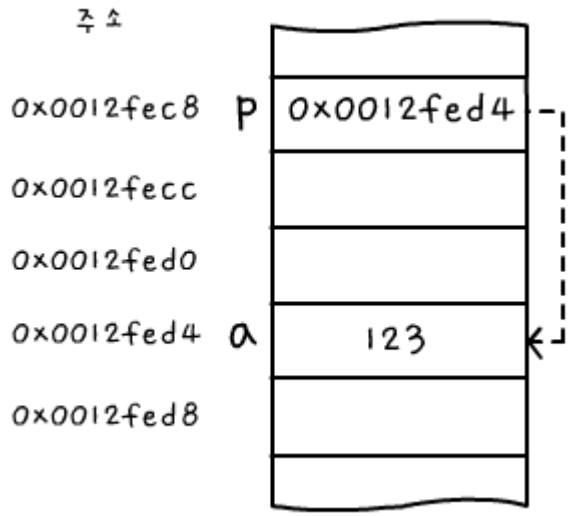
- 포인터 변수 - 다른 변수를 가리키는 변수
  - 포인터 변수에는 다른 변수의 주소 값을 저장할 수 있다.
  - 예) 포인터 변수 A가 다른 변수 B의 주소 값을 보관하고 있다면, 포인터 변수 A가 변수 B를 가리키고 있다고 말한다.



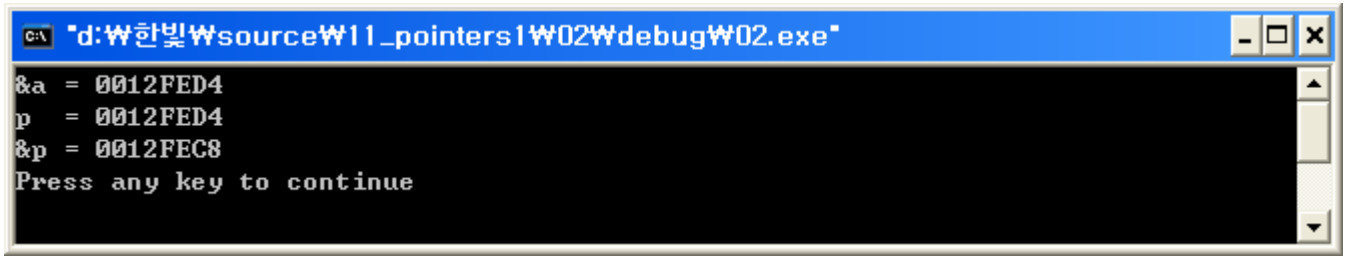
포인터 변수에 변수 주소 보관하기

- 포인터 변수 p가 변수 a를 가리키도록 만드는 예

```
// 일반적인 변수를 정의한다.
int a = 123;

// 포인터 변수를 정의한다.
int* p;

// p가 a를 가리키도록 만든다.
p = &a;

// 관련 정보를 출력한다.
cout << "&a = " << &a << "\n";
cout << "p  = " << p << "\n";
cout << "&p = " << &p << "\n";
```

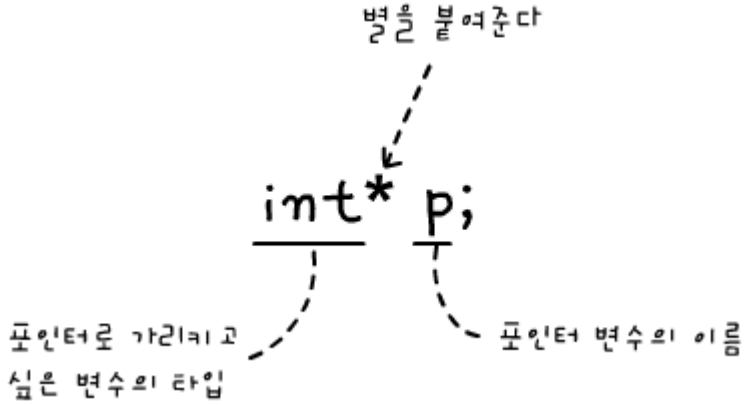


- 실행 결과



포인터 변수의 정의

- 포인터 변수를 정의할 때는 가리키고자 하는 변수의 타입을 지정해 두어야 한다.



- 다양한 타입의 포인터 변수

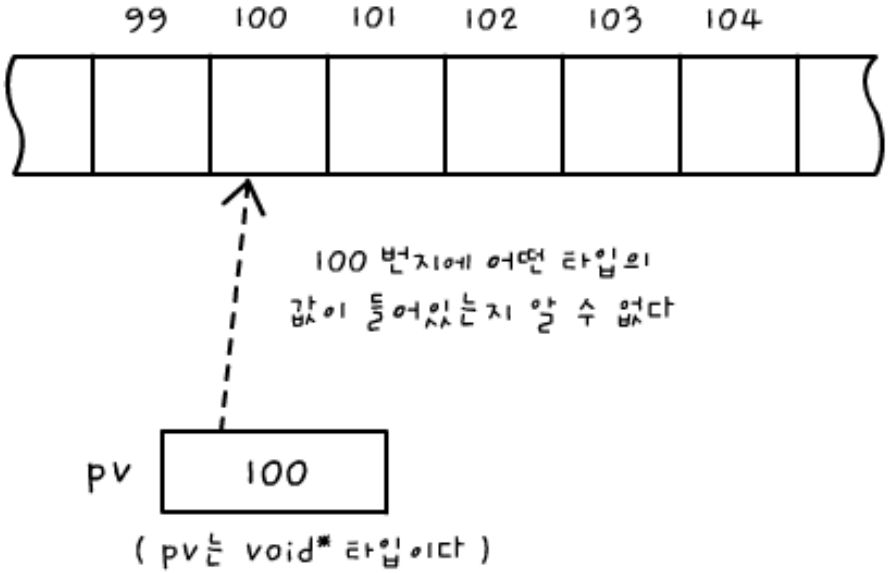
```
char c = 'C';
char* pc = &c;

float f = 700.5f;
float* pf = &f;

bool b = true;
bool* pb = &b;

short int s = 456;
short int* ps = &s;
```

void 포인터

```
void* p;
```

- void* 타입의 포인터 변수는 어떤 타입의 변수라도 가리킬 수 있음

- 그러므로, void 포인터가 가리키는 데이터의 크기나 종류를 알아낼 수 없음.



포인터 안전하게 사용하기

- 잘못된 주소를 가진 포인터를 사용하는 것은 매우 위험하기 때문에, 포인터를 사용할 때는 다음의 가이드 라인을 따른다.
  - 포인터 변수는 항상 0 혹은 NULL 값으로 초기화 한다.
  - 포인터 변수를 사용하기 전에는 0 혹은 NULL 값을 가지고 있는지 확인한다.

```
// 포인터 변수를 정의하고 초기화한다.
int* p = NULL;

// 이 상태에서 포인터를 사용해보자.
if (NULL != p)
        *p = 30;

// p가 변수를 가리키게 만들자
int a = 100;
p = &a;

// 이 상태에서 포인터를 사용해보자.
if (!p)
        *p = 30;
```

Const 속성을 가진 변수

- 변수의 값이 변경되는 것을 막기위해서 Const 속성을 사용해서 변수를 정의할 수 있다.

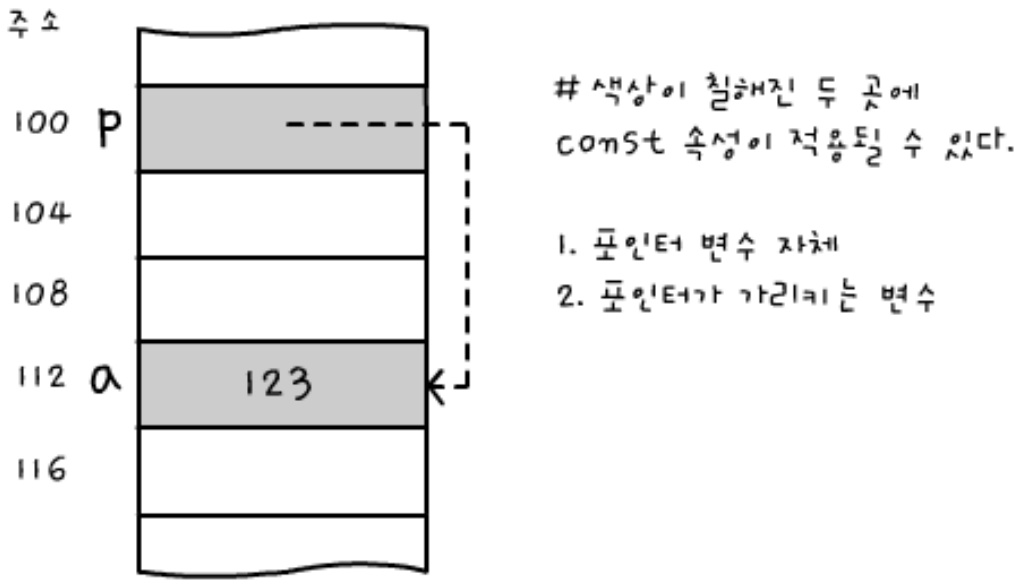
```
// const 속성을 가진 변수를 정의한다.
const int a = 123;

// a의 값을 바꾸려고 했으므로 컴파일 에러 발생!!
a = 456;
```

- 배열의 크기 값을 보관하기 위해서 Const 변수를 사용한 예

```
// 배열의 크기를 const 변수에 보관한다.
const unsigned int arraySize = 100;

// 배열을 정의한다.
char characters[ arraySize ] = {0};

// 배열을 사용한다.
for (int i = 0; i < arraySize; ++i)
        characters[i] = i + 1;
```

Const 와 포인터 (1)

- 포인터 변수에 Const 속성을 부여하는 경우
  - 포인터 변수 자체
  - 포인터 변수가 가리키는 변수



Const 와 포인터 (2)

- 포인터 변수에 Const 속성을 적용한 3가지 경우 비교
  - 포인터 변수가 가리키는 변수가 const인 경우

```
int i1 = 10;
int i2 = 20;
const int* p = &i1;

p = &i2;    // OK
*p = 30;    // FAIL
```

  - 포인터 변수 자신이 const인 경우

```
int i1 = 10;
int i2 = 20;
int* const p = &i1;

p = &i2;    // FAIL
*p = 30;    // OK
```

  - 두 변수 모두 const인 경우

```
int i1 = 10;
int i2 = 20;
const int* const p = &i1;

p = &i2;    // FAIL
*p = 30;    // FAIL
```

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;
```

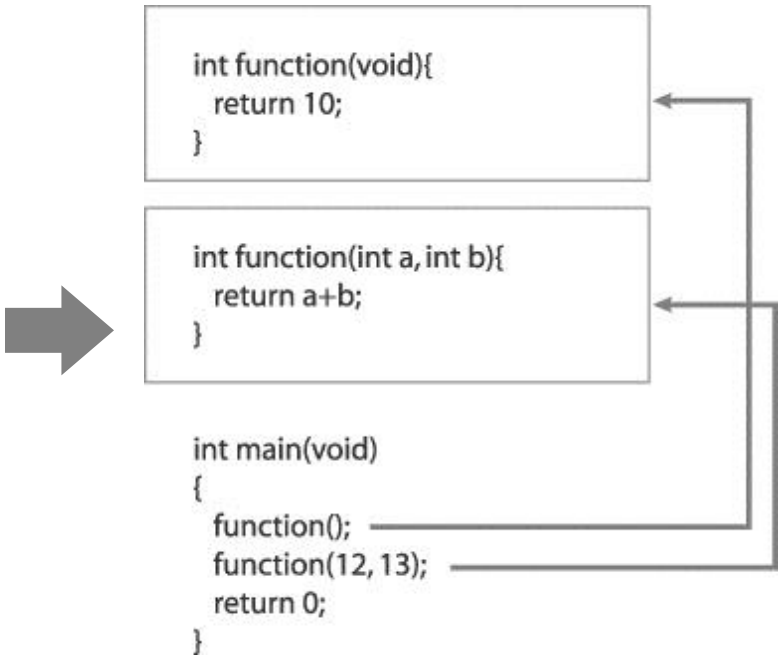
```
std::cin>>입력 변수1>>입력 변수 2>>입력 변수3;
```

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

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

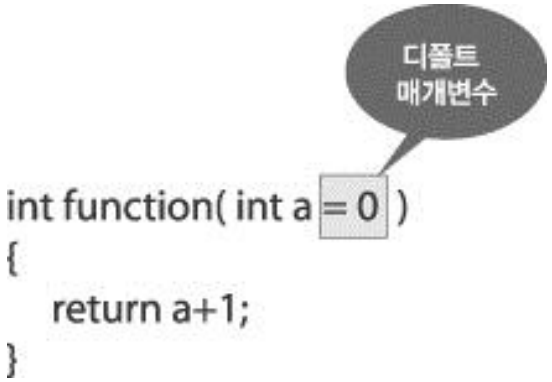
- **함수 오버로딩의 예**

```
int function1(int n){…}
int function1(char c){…}
```

```
int function2(int v){…}
int function2(int v1, int v2){…}
```

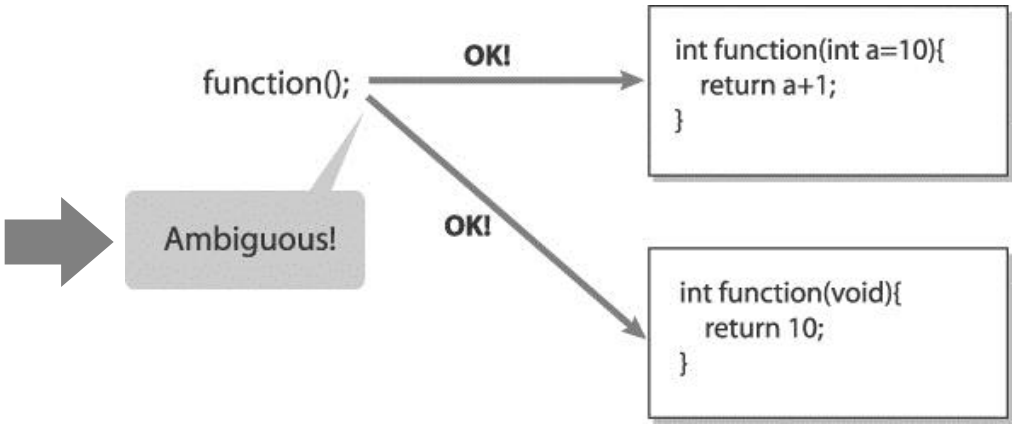
디폴트 매개 변수

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



- 디폴트 매개변수 vs. 함수 오버로딩

```
#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;
}
```



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

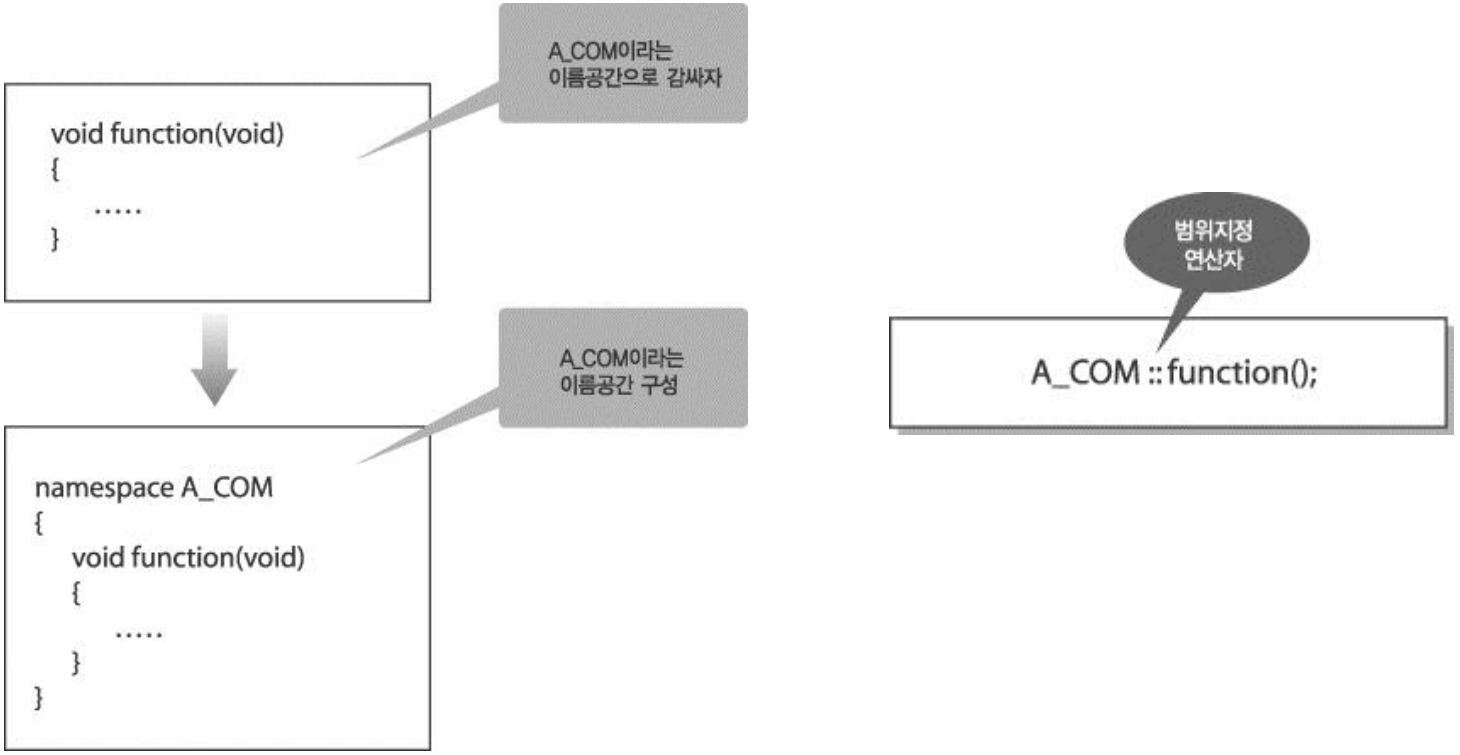
```
#include <iostream>
inline int SQUARE(int x)
{
   return x*x;
}

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;
}
```

- **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!

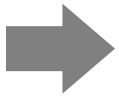
- **편의를 위한 using 선언!**

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

```
using namespace A_COM;
```

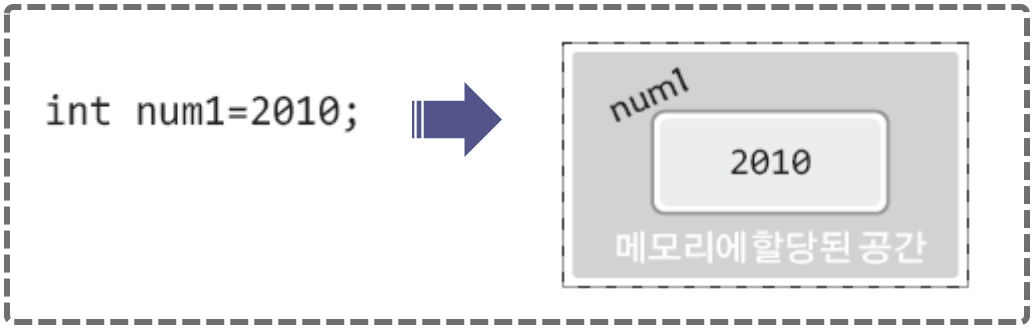
- **범위 지정 연산자 기반 전역 변수 접근**

```
int val=100;

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

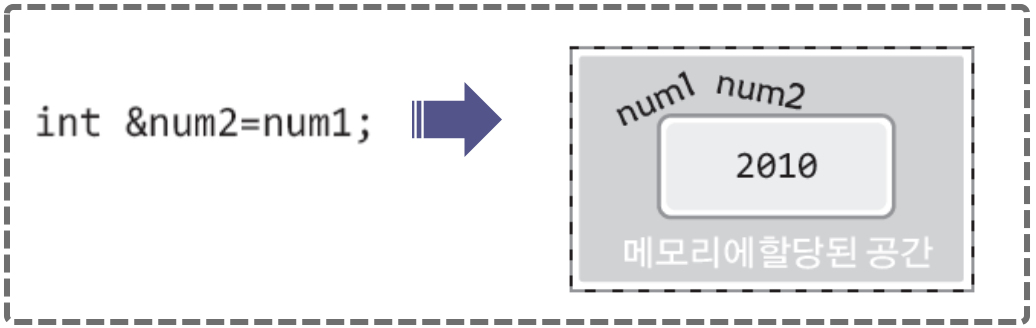


```
int val=100;

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

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

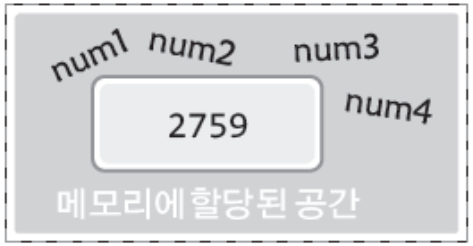
```
int main(void)
{
    int num1=1020;
    int &num2=num1;
    num2=3047;
    cout<<"VAL: "<<num1<<endl;
    cout<<"REF: "<<num2<<endl;
    cout<<"VAL: "<<&num1<<endl;
    cout<<"REF: "<<&num2<<endl;
    return 0;
}
```

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 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
```

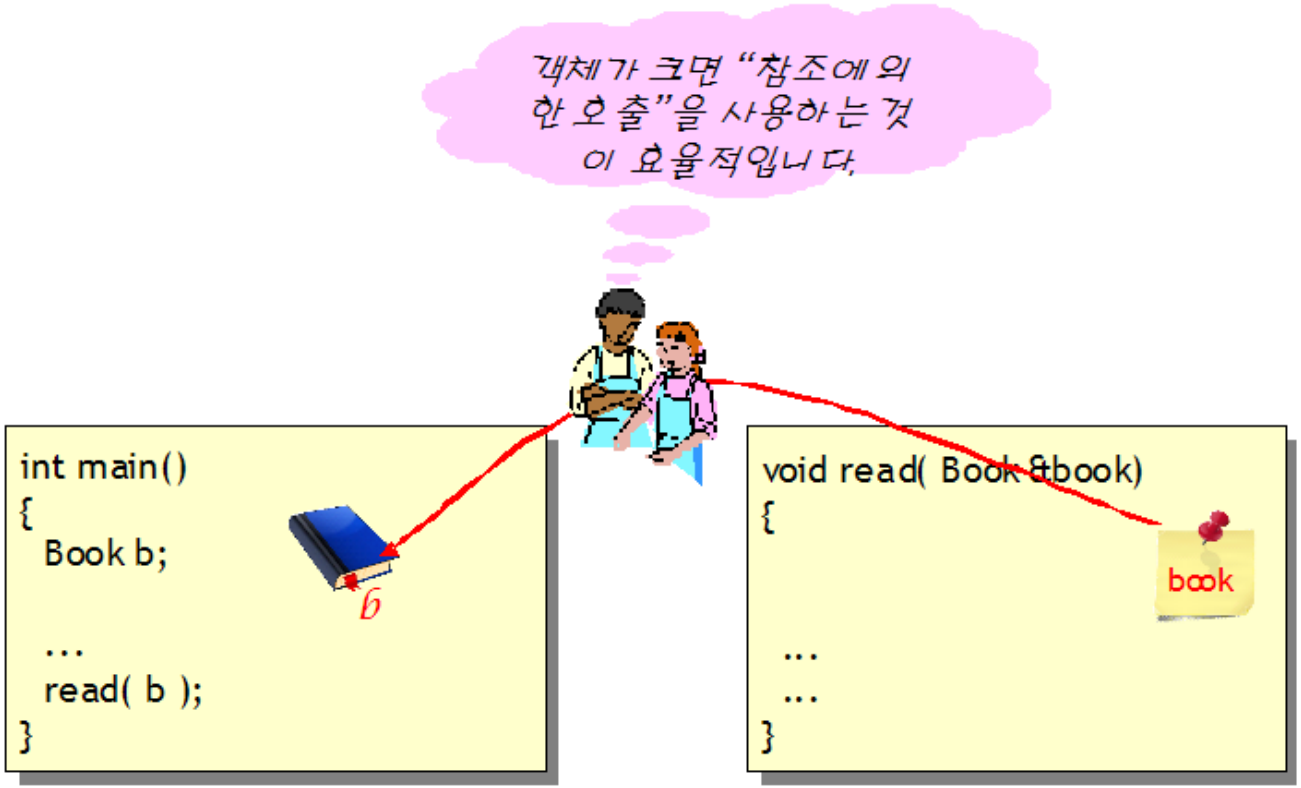
매개변수는 함수가 호출될 때 선언이 되는 변수이므로, 함수 호출의 과정에서 선언과 동시에 전달되는 대상으로 초기화된다.

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.

참고문헌

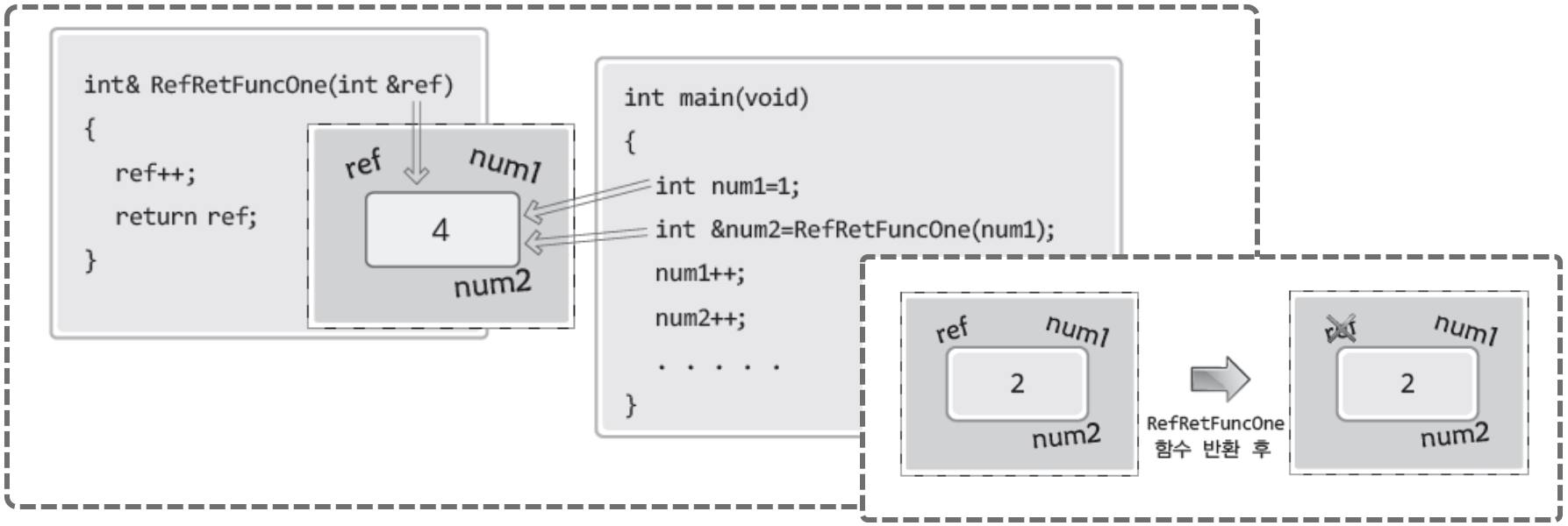
- 뇌를 자극하는 C++ 프로그래밍, 이현창, 한빛미디어, 2011
- 열혈 C++ 프로그래밍(개정판), 윤성우, 오렌지미디어, 2012
- 객체지향 원리로 이해하는 ABSOLUTE C++ , 최영근 외 4명 , 교보문고, 2013
- C++ ESPRESSO, 천인국, 인피니티북스, 2011
- 명품 C++ Programming, 황기태 , 생능출판사, 2013

Q&A

추가 참고자료

-참조자 부연 설명
-포인터사용 없이 힙에 접근 하기

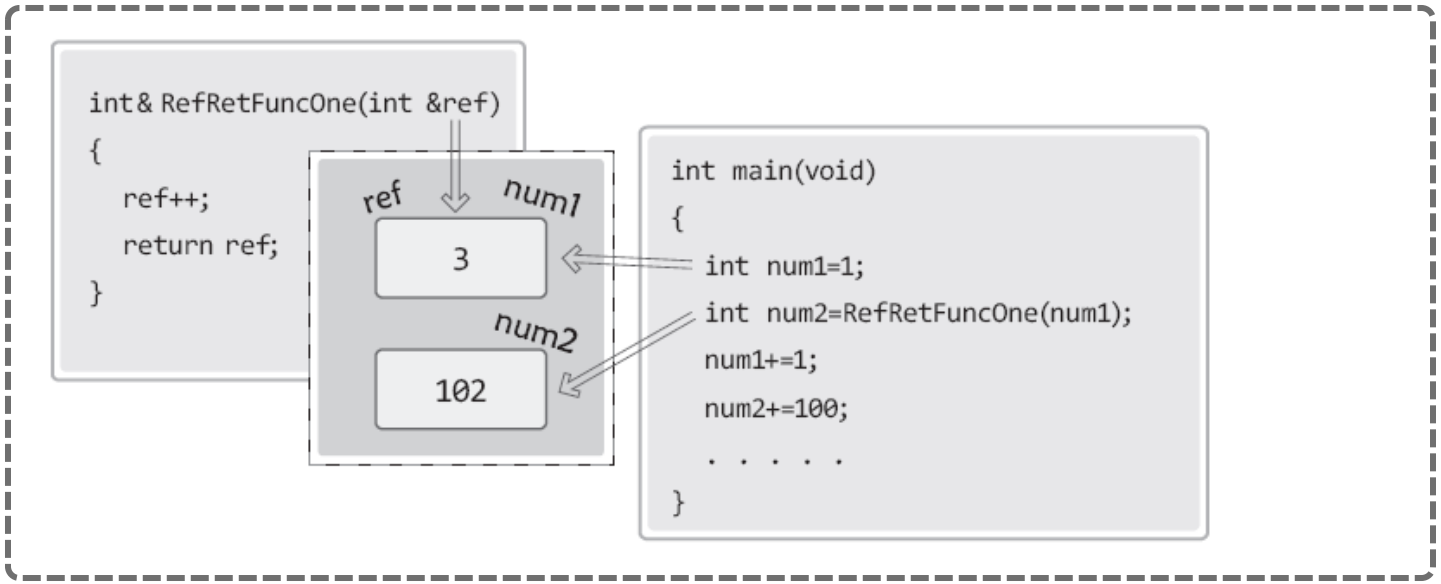
반환형이 참조이고 반환도 참조로 받는 경우



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!