C언어 기반의 C++

박 종 혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

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
```

'참'을 의미하는 true와 '거짓'을 의미하는 false

```
int main(void)
{
    int num=10;
    int i=0;
    cout<<"true: "<<true<<endl;
    cout<<"false: "<<false<<endl;
    while(true)
    {
        cout<<i++<<' ';
        if(i>num)
            break;
    }
    cout<<endl;
    cout<<"sizeof 1: "<<sizeof(1)<<endl;
    cout<<"sizeof 0: "<<sizeof(0)<<endl;
    cout<<"sizeof true: "<<sizeof(true)<<endl;
    cout<<"sizeof false: "<<sizeof(false)<<endl;
    return 0;
}
```

true는 '참'을 의미하는 1바이트 데이터이고, **false**는 '거짓'을 의미하는 1바이트 데이터이다. 이 둘은 각각 정수 1과 0이 아니다. 그러나 정수가 와야 할 위치에 오게 되면 , 각각 1과 0으로 변환이 된다.

```
• int num1=true;          // num1에는 1이 저장된다.
• int num2=false;         // num2에는 0이 저장된다.
• int num3=true+false;    // num3=1+0;
```

실행결과

```
true: 1
false: 0
0 1 2 3 4 5 6 7 8 9 10
sizeof 1: 4
sizeof 0: 4
sizeof true: 1
sizeof false: 1
```

자료형 bool

bool의 이해

- true와 false는 bool형 데이터이다.
- true와 false 정보를 저장할 수 있는 변수는 bool형 변수이다.

```
bool isTrueOne=true;
bool isTrueTwo=false;
```

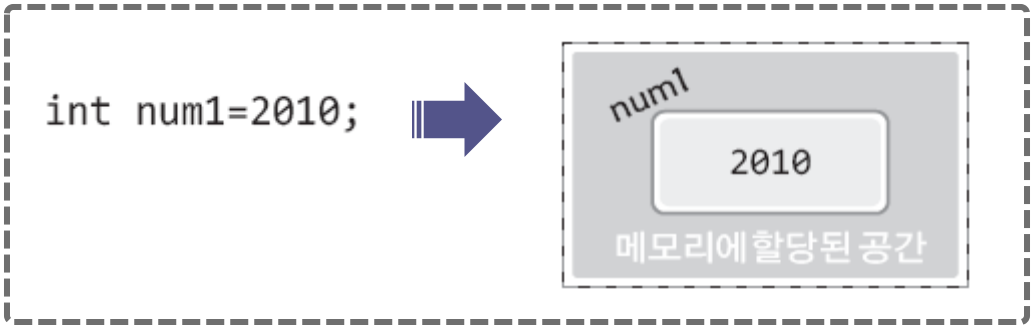
```
int main(void)
{
    bool isPos;
    int num;
    cout<<"Input number: ";
    cin>>num;
    isPos=IsPositive(num);
    if(isPos)
        cout<<"Positive number"<<endl;
    else
        cout<<"Negative number"<<endl;

    return 0;
}
```

실행결과

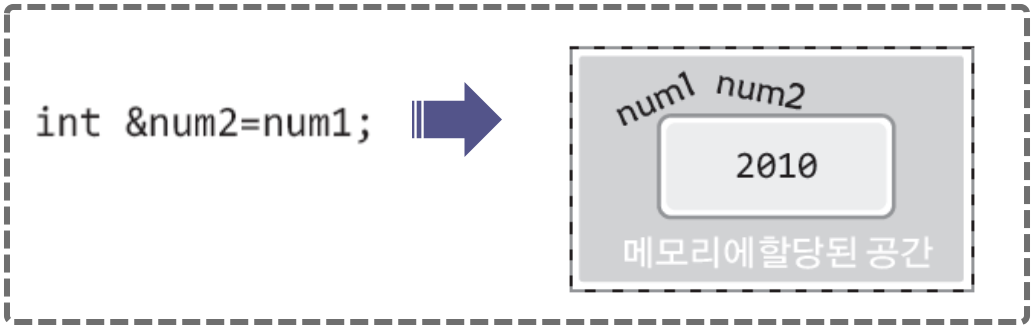
```
Input number: 12
Positive number
```

```
bool IsPositive(int num)
{
    if(num<0)
        return false;
    else
        return true;
}
```

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.

참조자 관련 예제와 참조자의 선언

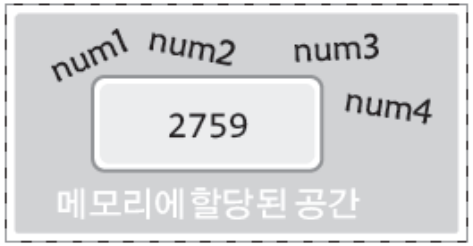
```
int main(void)
{
    int num1=1020;
    int &num2=num1;
    num2=3047;
    cout<<"VAL: "<<num1<<endl;
    cout<<"REF: "<<num2<<endl;
    cout<<"VAL: "<<&num1<<endl;
    cout<<"REF: "<<&num2<<endl;
    return 0;
}
```

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
```

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;
}
```

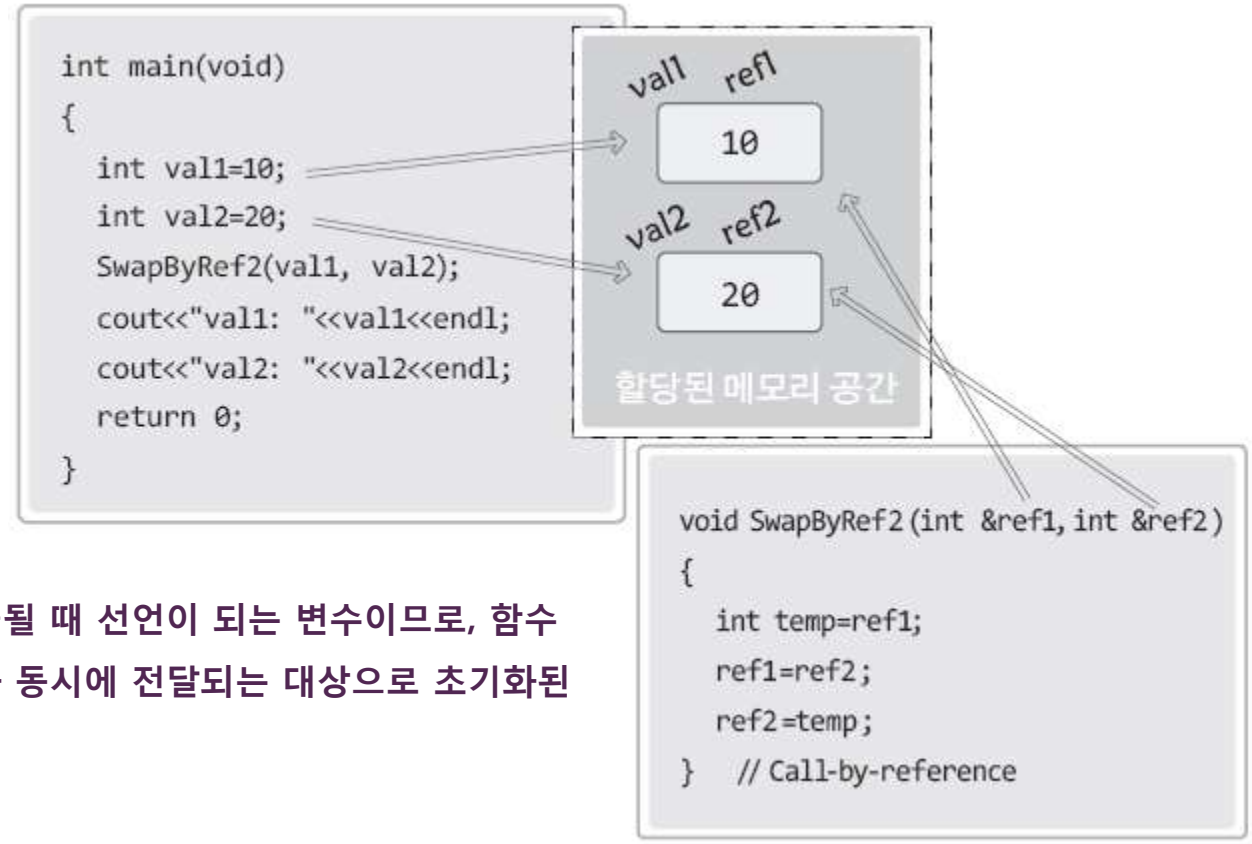
포인터 ptr에 전달된 주소 값의 관점에서 보면 이는 Call-by-value이다.

```
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.

참조자를 이용한 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!

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!

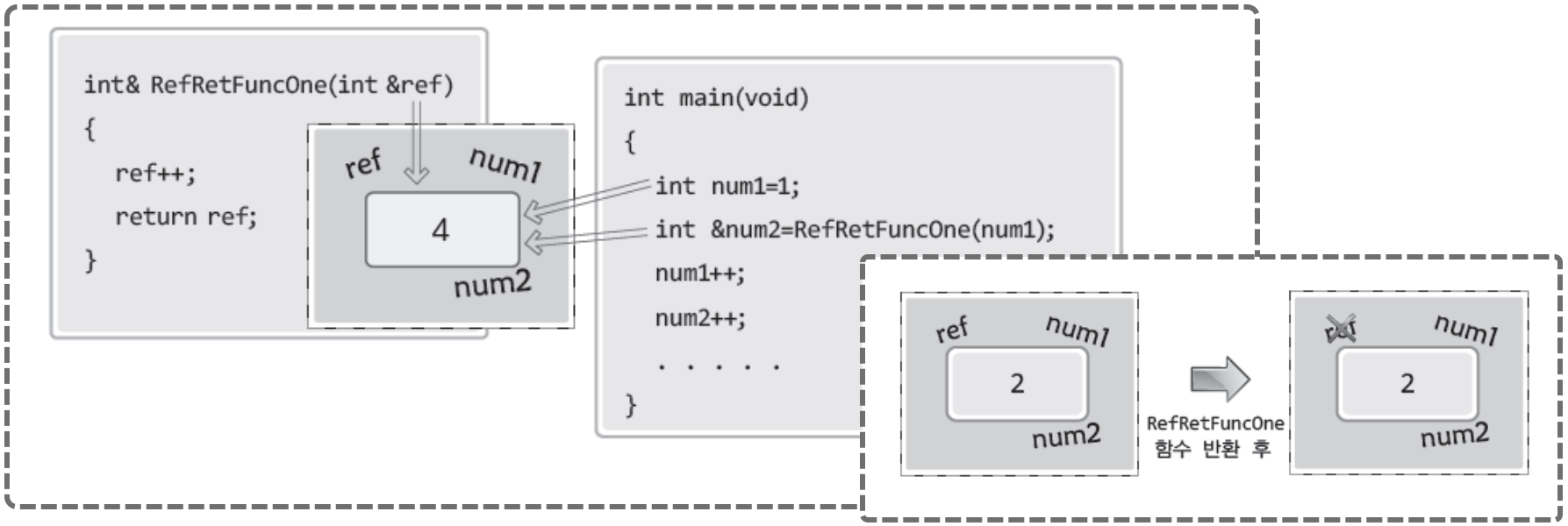
```
void HappyFunc(const int &ref) { . . . . }
```

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.

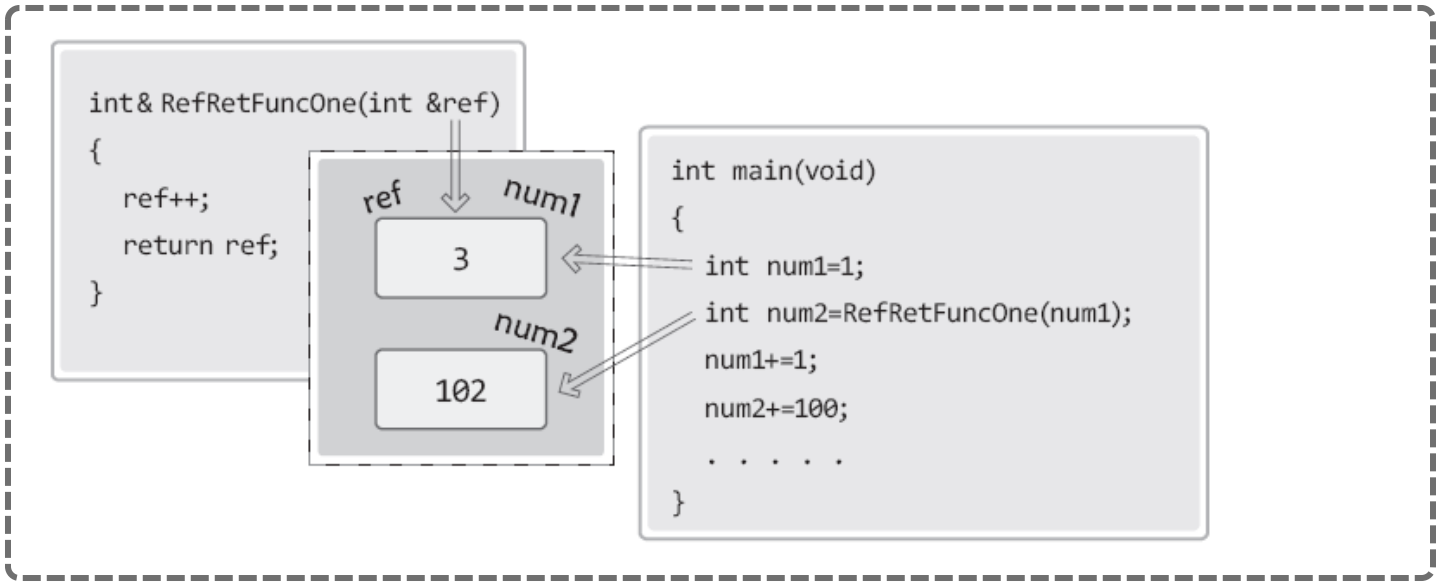
반환형이 참조이고 반환도 참조로 받는 경우



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 참조이되 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!

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!

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!

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.

Q&A