C++ 프로그래밍

김 형 기

hk.kim@jbnu.ac.kr

Today

- 포인터
- 참조자

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

포인터

(Summary) Pointers

- 포인터 개요
- 포인터의 정의
- 주소로의 접근
- 역참조
- 동적 메모리 할당
- 포인터와 배열
- 포인터와 const
- 포인터의 pass-by-reference
- 주의사항

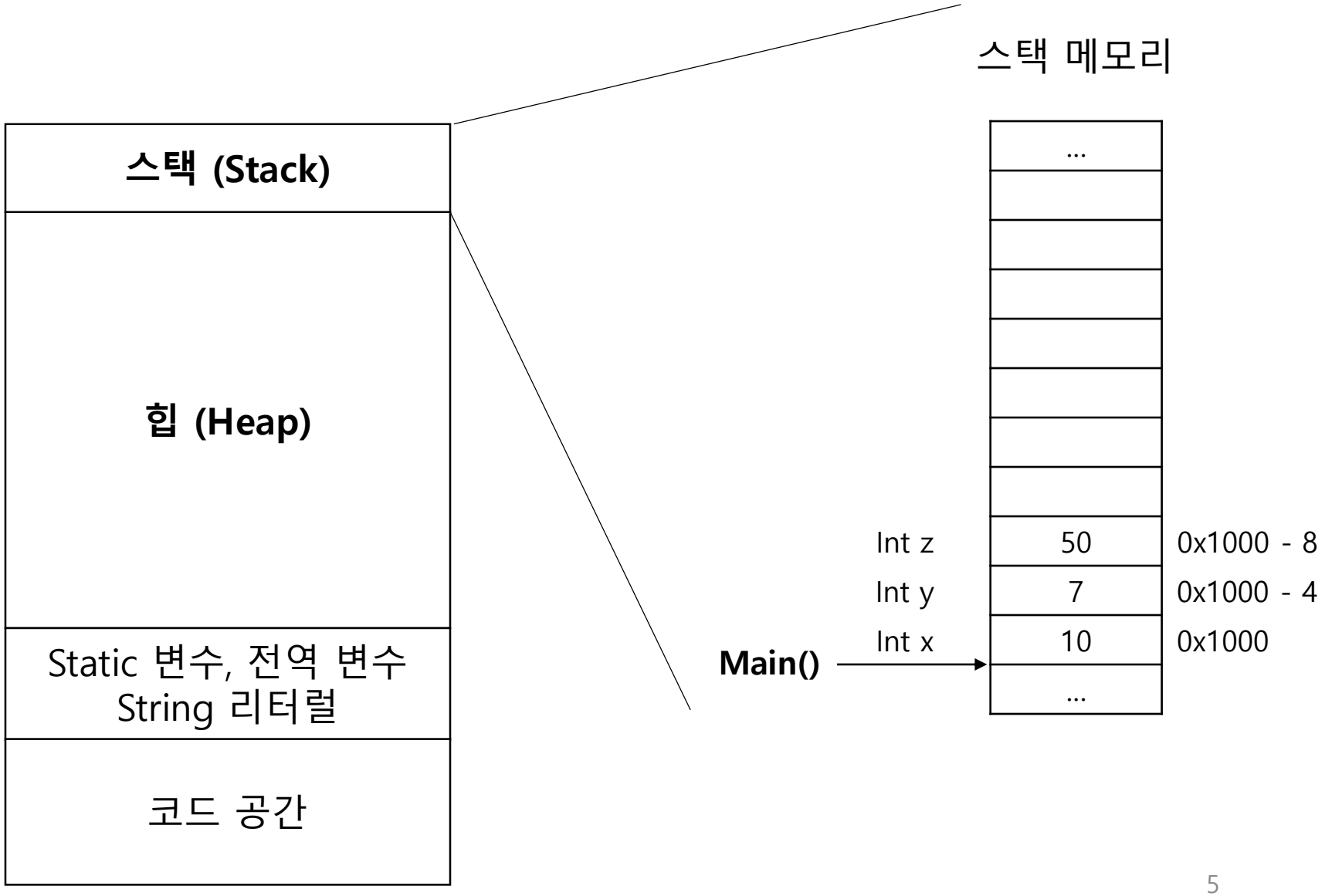
문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pointer

- 메모리

스택 메모리는 빠르지만 작음
메모리가 자동으로 정리(해제)됨

힙 메모리는 크지만 느림
메모리를 직접 해제해주어야 함



Pointer

고급

- 메모리

Stack vs Heap Pros and Cons

Stack

- very fast access
- don't have to explicitly de-allocate variables
- space is managed efficiently by CPU, memory will not become fragmented
- local variables only
- limit on stack size (OS-dependent)
- variables cannot be resized

Heap

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- you must manage memory (you're in charge of allocating and freeing variables)
- variables can be resized using `realloc()`

Pointers vs References

중요!

- 간단 정리
  - *의 사용
    - 변수를 정의할 때 붙는다? → 포인터의 정의(포인터 변수의 생성)
    - 변수를 사용할 때 붙는다? → 포인터의 역참조
  - &의 사용
    - 변수를 정의할 때 붙는다? → 참조자의 정의(참조자의 생성)
    - 변수를 사용할 때 붙는다? → 변수의 주소값 반환

```
int* a

*a

int& b

&a
```

Pointer

중요!

- 포인터 변수는 변수의 타입 중 하나
  - 변수의 타입: Int, float, double, int*, float*, …
    - Int와 int*는 다른 타입임!
  - 포인터 타입 변수가 가질 수 있는 값은 메모리의 **주소(memory address)**
    - 지금까지 사용한 변수와 포인터 변수와의 차이: 값을 저장하는지, 주소를 저장하는지
  - 포인터는 가리키는 주소에 저장된 데이터의 타입을 알아야 함

- 포인터를 쓰는 이유
  - **동적 할당을 통해 힙 영역의 메모리를 사용**
  - **변수의 범위 문제로 접근할 수 없는 곳의 데이터를 사용(참조자와 유사한 목적)**
  - 배열의 효율적인 사용
  - 다형성은 포인터를 기반으로 구현됨
  - 시스템 응용 프로그램 / 임베디드 프로그래밍에서는 메모리에 직접 접근이 필요함

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Definition of Pointers

- 포인터의 정의
  - 기존 변수 타입 뒤에 "*"를 붙여 포인터 변수 정의
    - "int*"까지를 타입으로 생각

```
variableType* pointerName
```

- 포인터의 정의 및 초기화 방법

```
variableType* pointerName = nullptr;
```

```
int* intPtr = nullptr;
double* doublePtr = nullptr;
```

  - 초기화를 하지 않으면 쓰레기 값이 들어있는 상태이므로 방지가 필요
  - Nullptr은 "nowhere" 개념
    - 임의의 메모리 주소를 가리키고 있는 상태가 아니라, 아무것도 가리키지 않는 상태를 의미

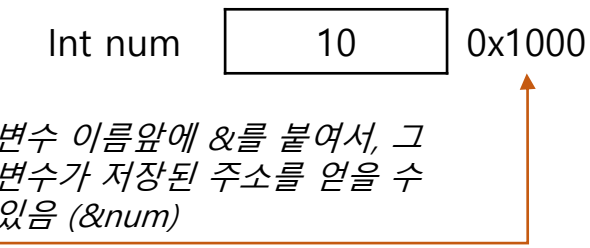
문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Accessing Pointer Address

중요!

- 변수의 주소값 얻어오기
  - 포인터 변수는 주소값을 저장하므로, 주소값을 얻어올 수 있어야 함
  - 이를 위해 주소 연산자("&")를 사용
    - 연산자가 적용되는 피연산자의 주소값이 반환됨, 즉 변수의 주소값이 반환됨
    - 피연산자는 주소값을 얻을 수 있는 종류여야 함(l-value)

```
1 int num=10;
2
3 cout << "Value : " << num << endl; //10
4 cout << "Address : " << &num << endl; //0x1000
5 cout << "Address : " << &10 << endl; //ERROR! 10은 주소가 없음
```



문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Accessing Pointer Address

- 변수의 주소값 얻어오기
  - 포인터 변수는 주소값을 저장하므로, 주소값을 얻어올 수 있어야 함
  - 이를 위해 주소 연산자(“&”)를 사용
    - 연산자가 적용되는 피연산자의 주소값이 반환됨, 즉 변수의 주소값이 반환됨
    - 피연산자는 주소값을 얻을 수 있는 종류여야 함(l-value)

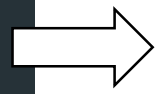
```
1 int* p;
2
3 cout << "Value : " << p << endl; // 0x66ff60 - garbage
4 cout << "Address : " << &p << endl; // 0x61ff18
5 cout << "Size : " << sizeof(p) << endl; //4 for address
6
7 p = nullptr;
8
9 cout << "Value : " << p << endl; // 0
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Accessing Pointer Address

- 주소값의 이해
  - "포인터 변수의 크기"와 "포인터가 가리키고 있는 대상의 크기"는 별개임!
  - 포인터 변수들은 모두 같은 크기
    - X86에서는 4바이트
    - 포인터는 "주소값"을 저장하기 때문
  - 타입은 왜 필요할까?
    - 해당 주소의 값에 접근할 때 몇 바이트 크기인지 알아야 함!

```
1 int* p1 = nullptr;
2 double* p2 = nullptr;
3 unsigned long long* p3 = nullptr;
4 vector<string>* p4 = nullptr;
5 string* p5 = nullptr;
```



주소에 저장된 변수의 메모리 크기는 모두 다르지만 포인터 변수가 저장하는 주소값은 모두 4바이트 크기로 동일함

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Accessing Pointer Address

- 포인터의 타입
  - 컴파일러는 포인터가 가리키는 타입이 맞는지 확인함
    - ➢ int*는 int가 저장된 주소만, double*는 double이 저장된 주소만 가리킬 수 있음

```
1 int score = 100;
2 double preciseScore = 100.7;
3
4 int* scorePtr = nullptr;
5 scorePtr = &score;
6 scorePtr = &preciseScore; //COMPILER ERROR!
```

Accessing Pointer Address

고급

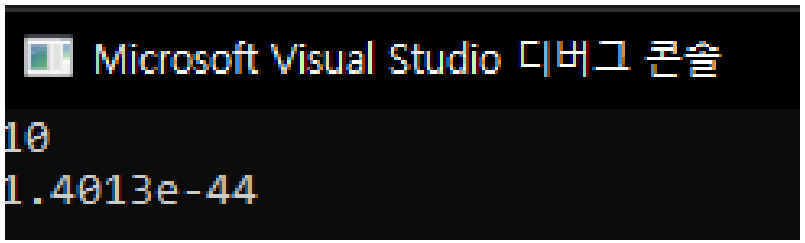
- 주소값의 이해
  - 타입은 왜 필요할까?
    - ➢ 해당 주소의 값에 접근할 때 몇 바이트 크기인지 알아야 함!

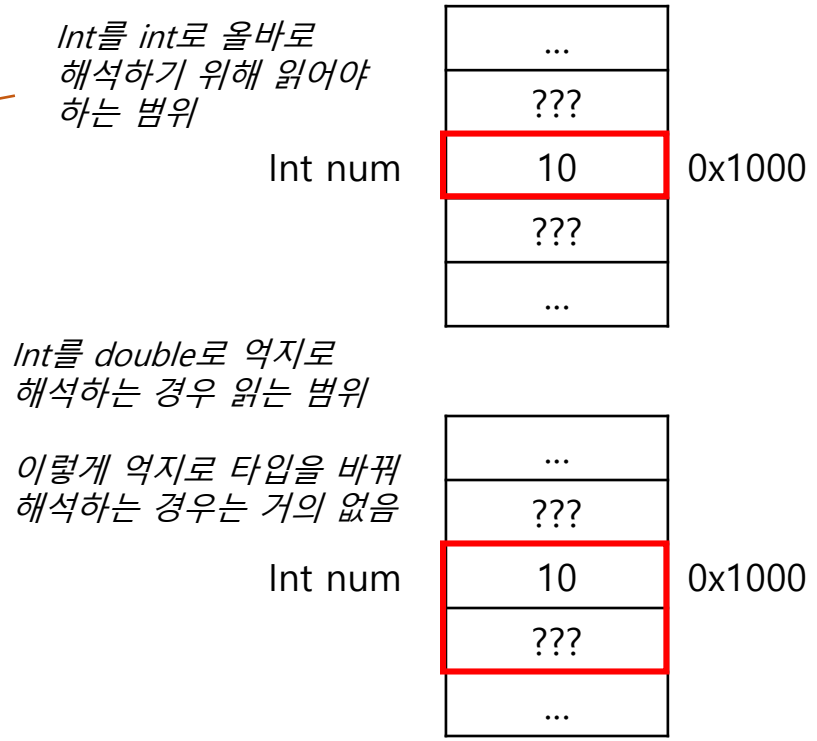
```cpp
int main()
{
    int num = 10;

    int* numIntPtr = &num;

    float* numDoublePtr = (double*)(void*)&num;

    std::cout << *numIntPtr << std::endl;
    std::cout << *numDoublePtr << std::endl;
}
```





Dereferencing a Pointer

중요!

- 포인터의 역참조
  - 포인터의 주소에 저장된 데이터에 접근
  - * 연산자를 사용

```
1 int score=10;
2 int* scorePtr = &score; //여기서의 *는 역참조 용도가 아님!!
3
4 cout << *scorePtr << endl; //10
5
6 *scorePtr = 20;
7 cout << *scorePtr << endl; //20
8 cout << score << endl;  //20
```

Dereferencing a Pointer

- 포인터의 역참조
  - 포인터의 주소에 저장된 데이터에 접근
  - * 연산자를 사용

```
1 double highTemp = 100.7;
2 double lowTemp = 37.4;
3 double* tempPtr = &highTemp;
4
5 cout << *tempPtr << endl;        // 100.7
6 temp_ptr = &lowTemp;
7 cout << *tempPtr << endl;        // 37.4
```

Memory Debugging

고급

- 포인터에 대한 메모리 디버깅
  - 메모리 창에서 저장된 변수를 볼 때는 &a 와 같이 변수 이름 앞에 &를 붙여 주었음
    - 왜 그랬는지 생각해 보세요.
  - 포인터가 가리키는 값을 보기위해서는 &가 필요 없겠죠?
    - 왜 그러할지 생각해 보세요.
  - 포인터 변수가 저장된 위치를 보기 위해서는 &가 필요하겠죠?
    - 왜 그러할지 생각해 보세요.

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Dynamic Memory Allocation

중요!

- 동적 메모리 할당
  - 런타임에 **"힙 메모리"를 할당**
    - ➢ 프로그램의 실행 도중 얼마나 많은 메모리가 필요한지 미리 알 수 없는 경우 사용
      - 사용자 입력에 따라 크기가 바뀌는 경우
      - 파일을 선택하여 내용을 읽어오는 경우 등
    - ➢ 큰 데이터를 저장해야 할 경우(stack은 크기가 작음, 몇 MB정도)
    - ➢ 객체의 생애주기(언제 메모리가 할당되고 해제되어야 할지)를 직접 제어해야 할 경우
  - **힙 메모리는 스택과는 달리 스스로 해제되지 않음!**
  - **사용이 끝나고 해제하지 않으면 메모리 누수 발생!**

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Dynamic Memory Allocation

중요!

- 동적 메모리 할당 방법
  - new 연산자 사용

```
1  int* intPtr=nullptr;
2
3  intPtr = new int; //allocate integer in heap
4
5  cout << intPtr << endl; // 0x2345f45
6
7  cout << *intPtr << endl; // garbage value
8
9  *intPtr = 100;
10
11 cout << *intPtr << endl; // 100
```

new의 역할(C의 malloc)

Heap 메모리 공간에 int 하나를 담을 수 있는 메모리 공간을 할당받은 뒤 그 주소값을 리턴

5장에서 배울 클래스 객체의 경우 생성자 호출까지 수행

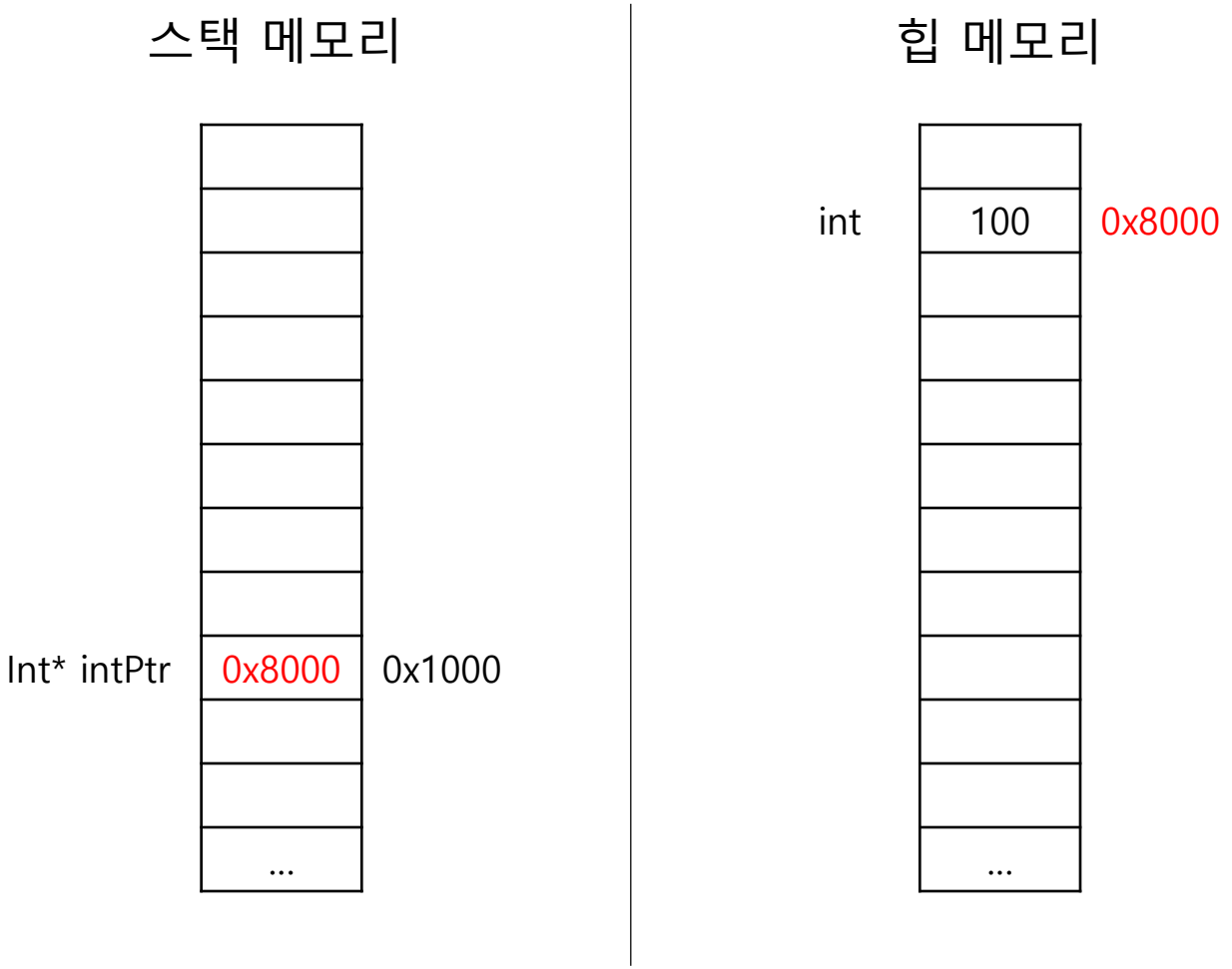
문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Dynamic Memory Allocation

중요!

- 동적 메모리 할당
  - new 연산자 사용

```
 1 int* intPtr=nullptr;
 2
 3 intPtr = new int; //allocate integer in heap
 4
 5 cout << intPtr << endl; // 0x2345f45
 6
 7 cout << *intPtr << endl; // garbage value
 8
 9 *intPtr = 100;
10
11 cout << *intPtr << endl; // 100
```



모든 명령문의 실행이 끝나고 난 후 메모리의 상태

(0x1000, 0x8000 등은 특별한 의미를 갖는것이 아니고,
그냥 "임의의 메모리 주소" 를 표현하는 숫자임)

Dynamic Memory Allocation

중요!

- 동적 메모리의 해제
  - delete 연산자 사용

```
1  int* intPtr=nullptr;
2
3  intPtr = new int;
4
5  cout << intPtr << endl;
6
7  cout << *intPtr << endl;
8
9  *intPtr = 100;
10
11 cout << *intPtr << endl;
12
13 delete intPtr; // free
14 intPtr=nullptr; //optional
```

delete의 역할(C의 free)

해당 메모리 공간의 데이터를
삭제하고 반납

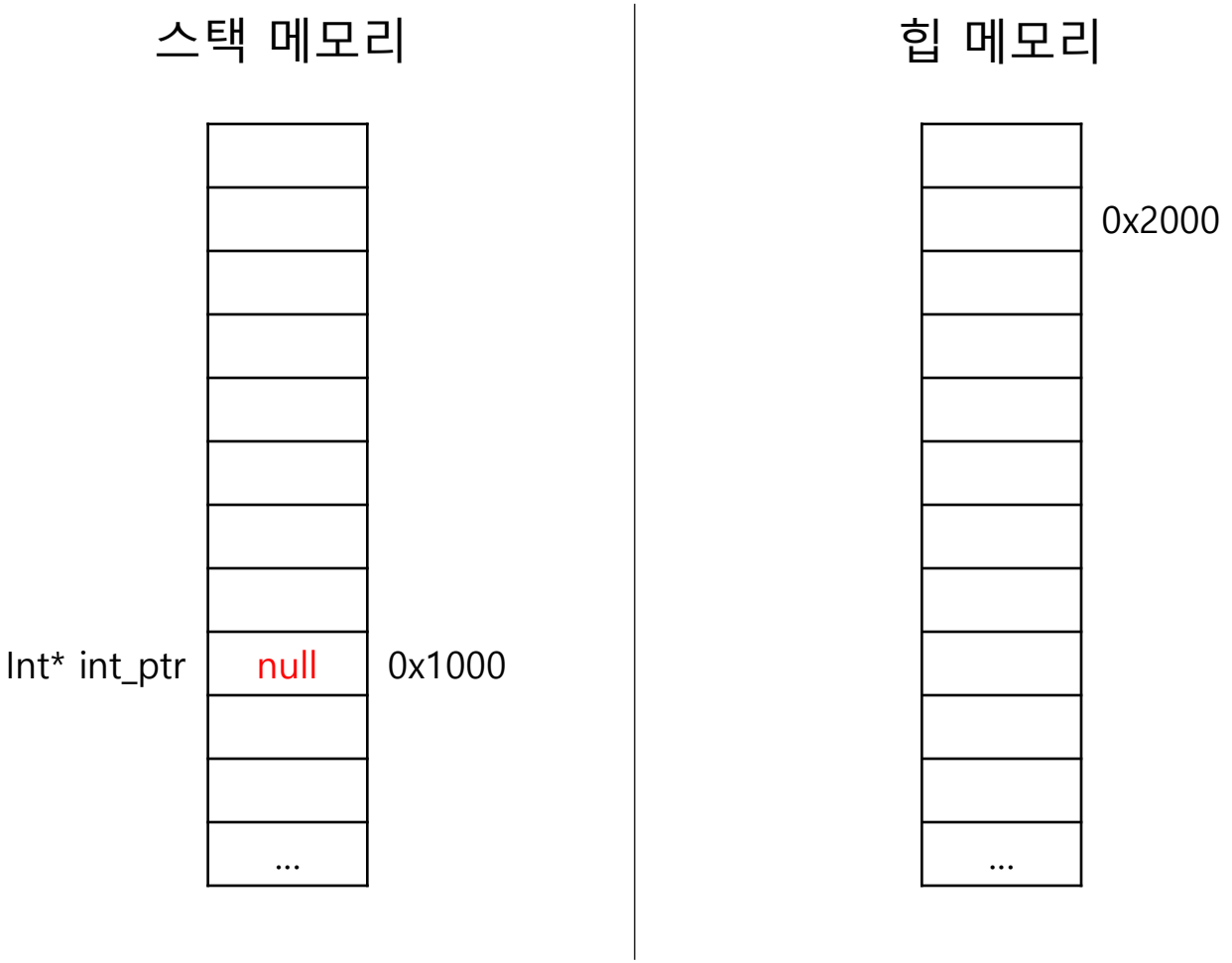
5장에서 배울 클래스 객체의 경우 소멸자 호출까지 수행

Dynamic Memory Allocation

중요!

- 동적 메모리의 해제
  - delete 연산자 사용

```
1  int* intPtr=nullptr;
2
3  intPtr = new int;
4
5  cout << intPtr << endl;
6
7  cout << *intPtr << endl;
8
9  *intPtr = 100;
10
11 cout << *intPtr << endl;
12
13 delete intPtr; // free
14 intPtr=nullptr; //optional
```



모든 명령문의 실행이 끝나고 난 후 메모리의 상태

Dynamic Memory Allocation

중요!

- 동적 할당을 이용한 배열
  - new[], delete[] 연산자 사용

```
1  int* arrayPtr = nullptr;
2  int size = 0;
3
4  cout << "size of array?";
5  cin >> size;
6
7  arrayPtr = new int[size];
8
9  arrayPtr[0] = 10;
10 arrayPtr[1] = 20;
11 arrayPtr[2] = 30;
12
13 delete[] arrayPtr;
```

New[] 의 역할(C의 malloc)

Heap 메모리 공간에 int를 size개수만큼 담을 수 있는 메모리 공간을 할당받은 뒤 주소값을 반환
5장에서 배울 클래스 객체의 경우 생성자 호출까지 수행

Delete[] 의 역할(C의 free)

해당 메모리 공간을 반납
5장에서 배울 클래스 객체의 경우 소멸자 호출까지 수행

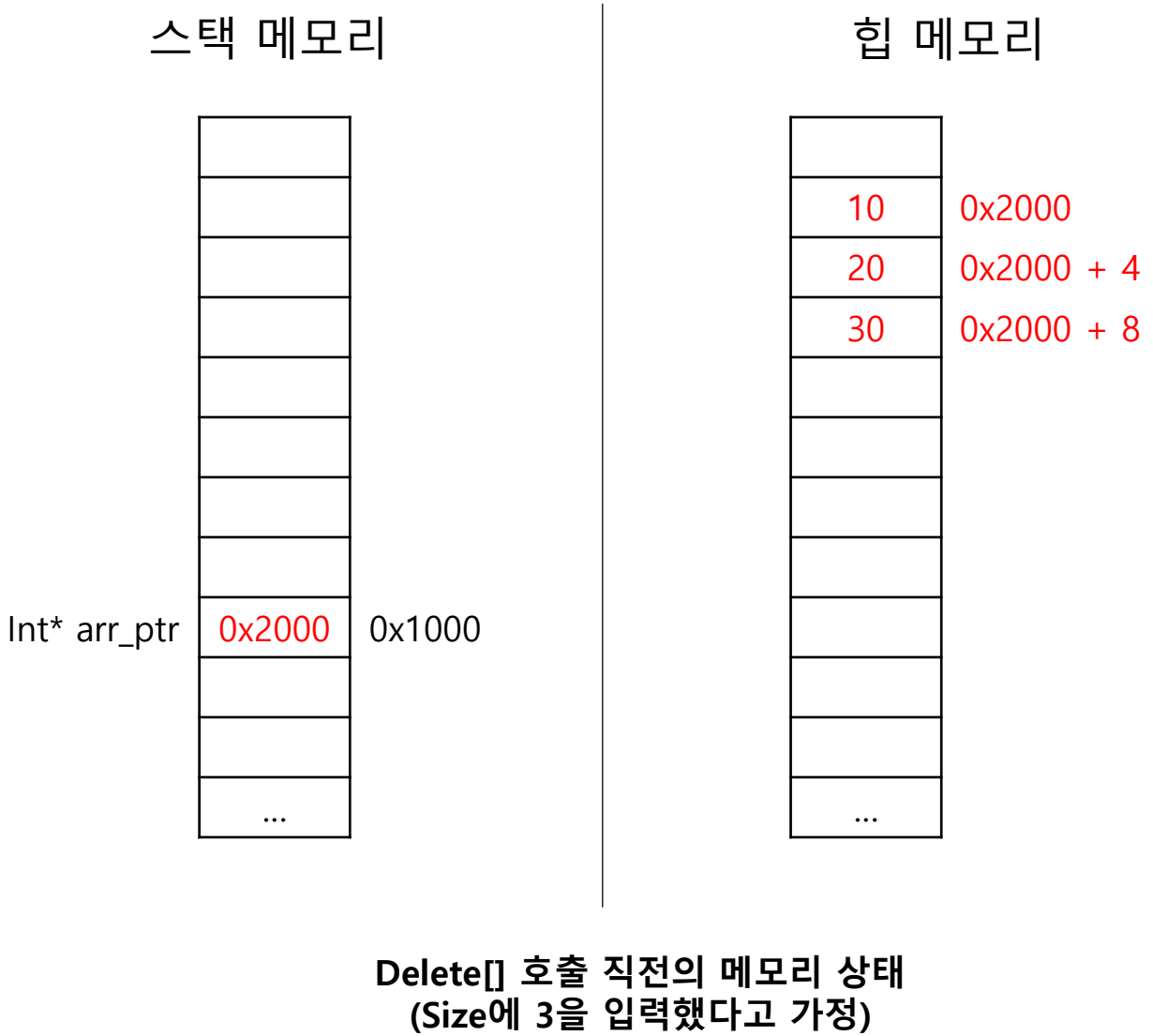
문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Dynamic Memory Allocation

중요!

- 동적 할당을 이용한 배열
  - new[], delete[] 키워드 사용

```
1  int* arrayPtr = nullptr;
2  int size = 0;
3
4  cout << "size of array?";
5  cin >> size;
6
7  arrayPtr = new int[size];
8
9  arrayPtr[0] = 10;
10 arrayPtr[1] = 20;
11 arrayPtr[2] = 30;
12
13 delete[] arrayPtr;
```



Delete[] 호출 직전의 메모리 상태
(Size에 3을 입력했다고 가정)

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Dynamic Memory Allocation

중요!

- 동적 할당을 이용한 배열
  - new[], delete[] 키워드 사용

```
1  int* arrayPtr = nullptr;
2  int size = 0;
3
4  cout << "size of array?";
5  cin >> size;
6
7  arrayPtr = new int[size];
8
9  arrayPtr[0] = 10;
10 arrayPtr[1] = 20;
11 arrayPtr[2] = 30;
12
13 delete[] arrayPtr;
```

Delete[] 호출 후
(Nullptr 대입해주지 않으면 문제가 생길지도?)

Arrays and Pointers

- 배열의 이름은 배열의 첫 요소의 주소를 가리킨다(remind)
- 포인터 변수의 값은 주소값이다
- 포인터 변수와 배열이 같은 주소를 가리킨다면, 포인터 변수와 배열은 (거의)동일하게 사용 가능하다
  - (차이점: 배열은 주소값을 정의 이후 변경 불가. Sizeof() 반환값이 다름)

```
1 int scores[] = {100,95,90};
2 cout << scores << endl; // 00F3FAF8
3 cout << *scores << endl; // 100
4
5 int* scorePtr = scores;
6 cout << scorePtr << endl; // 00F3FAF8
7 cout << *scorePtr << endl; // 100
```

Arrays and Pointers

```
1 int arrayName[] = {1,2,3,4,5};
2 int* pointerName = arrayName;
```

- arrayName[index] == pointerName[index]
- *(arrayName + index) == *(pointerName + index)

```
 1 int scores[] = { 100,95,90 };
 2 int* scoresPtr = scores;
 3
 4 cout << scoresPtr << endl; // 0x123400
 5 cout << (scoresPtr + 1) << endl; // 0x123404
 6 cout << (scoresPtr + 2) << endl; // 0x123408
 7
 8 cout << *scoresPtr << endl; // 100
 9 cout << *(scoresPtr + 1) << endl; // 95
10 cout << *(scoresPtr + 2) << endl; // 90
```

Const and Pointers

1. const의 포인터(pointers to const)
2. const인 포인터(const pointers)
3. const의 const인 포인터(const pointers to const)

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Const and Pointers

1. const의 포인터(pointers to const)

- 데이터가 const / 포인터는 다른 데이터를 가리킬 수 있음

2. const인 포인터(const pointers)

3. const의 const인 포인터(const pointers to const)

```
1 int highScore = 100;
2 int lowScore = 60;
3 const int* scorePtr = &highScore;
4
5 *scorePtr = 80; // ERROR
6 scorePtr = &lowScore; // OK
```

Const and Pointers

1. const의 포인터(pointers to const)
2. **const인 포인터(const pointers)**
   - **포인터가 const / 데이터는 변할 수 있음**
3. const의 const인 포인터(const pointers to const)

```
1 int highScore = 100;
2 int lowScore = 60;
3 int* const scorePtr =&highScore;
4
5 *scorePtr = 80; // OK
6 scorePtr = &lowScore; // ERROR
```

Const and Pointers

1. const의 포인터(pointers to const)
2. const인 포인터(const pointers)
3. **const의 const인 포인터(const pointers to const)**
   - **둘 다 const**

```
1 int highScore = 100;
2 int lowScore = 60;
3 const int* const scorePtr = &highScore;
4
5 *scorePtr = 80; // ERROR
6 scorePtr = &lowScore; // ERROR
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Passing Pointers to a Function

중요!

- 포인터를 함수의 인자로 전달
  - Pass-by-address / 변수의 주소를 전달

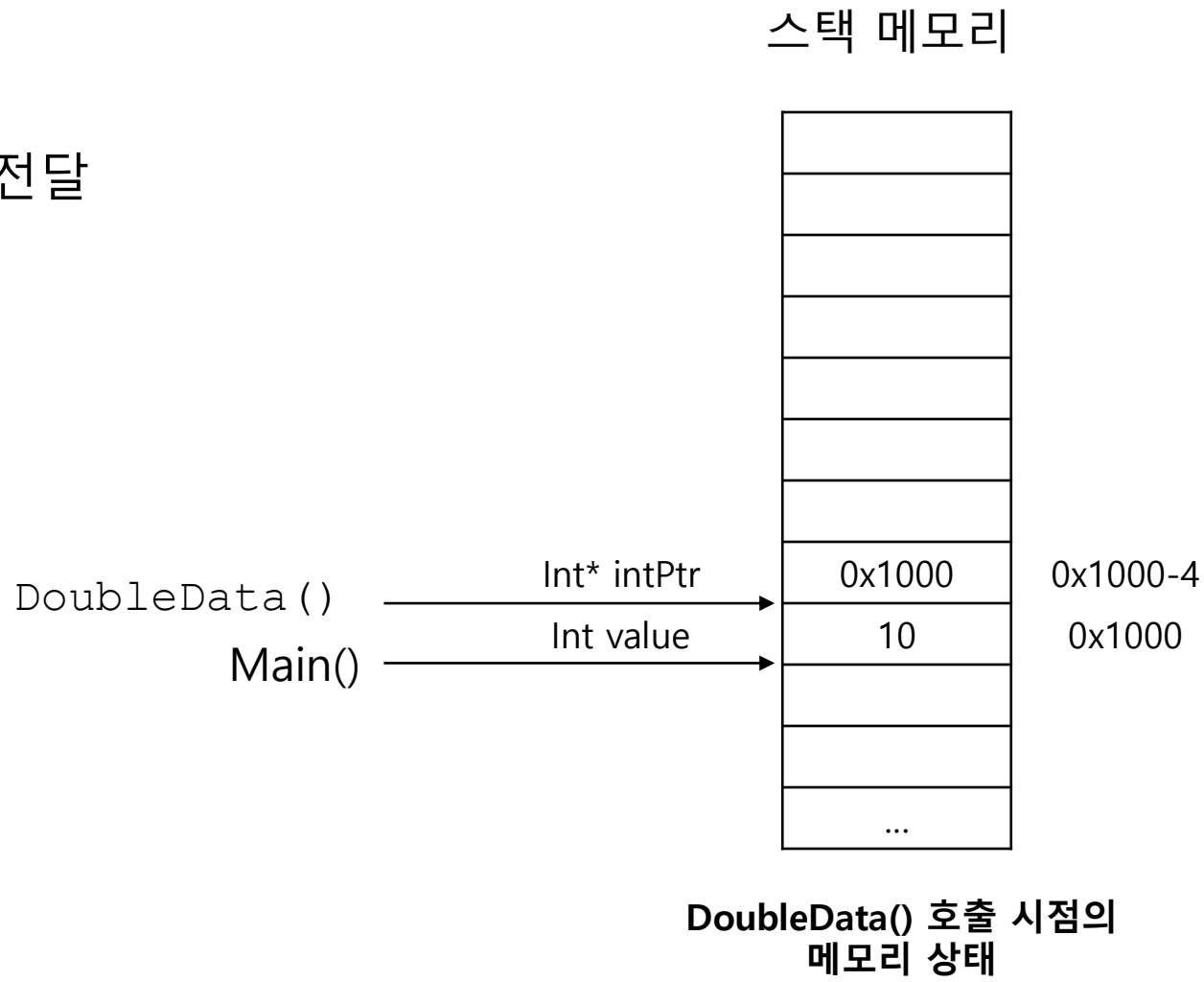
```
1  void DoubleData(int* intPtr)
2  {
3      *intPtr *= 2;
4  }
5
6  int main()
7  {
8      int value = 10;
9
10     cout << value << endl; // 10
11
12     DoubleData(&value);
13
14     cout << value << endl; //20
15 }
```

Passing Pointers to a Function

중요!

- 포인터를 함수의 인자로 전달
  - Pass-by-address / 변수의 주소를 전달

```
1  void DoubleData(int* intPtr)
2  {
3      *intPtr *= 2;
4  }
5
6  int main()
7  {
8      int value = 10;
9
10     cout << value << endl; // 10
11
12     DoubleData(&value);
13
14     cout << value << endl; //20
15 }
```



DoubleData() 호출 시점의 메모리 상태

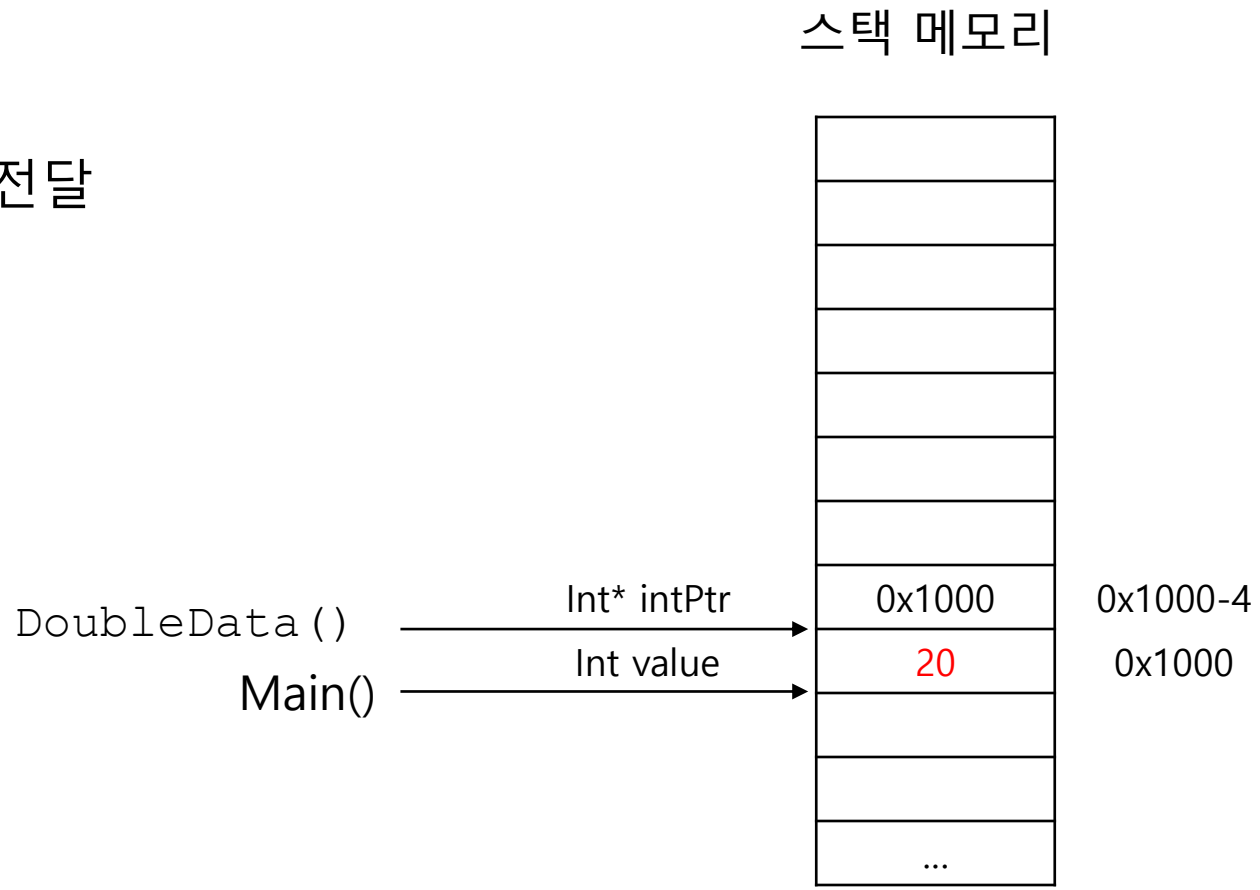
문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Passing Pointers to a Function

중요!

- 포인터를 함수의 인자로 전달
  - Pass-by-address / 변수의 주소를 전달

```
1  void DoubleData(int* intPtr)
2  {
3      *intPtr *= 2;
4  }
5
6  int main()
7  {
8      int value = 10;
9
10     cout << value << endl; // 10
11
12     DoubleData(&value);
13
14     cout << value << endl; //20
15 }
```



DoubleData() 호출 종료 시점의 메모리 상태

주소를 가지고 있으니 호출 Stack 밖의 값도 바꿀 수 있다!

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Returning a Pointer

- 포인터의 반환
  - **인자로 전달된 데이터(포인터)를 반환 → OK**

```
1 int* LargerInt(int* intPtr1, int* intPtr2)
2 {
3     if(*intPtr1 > *intPtr2)
4         return intPtr1;
5     else
6         return intPtr2;
7 }
```

Returning a Pointer

- 포인터의 반환
  - **함수 내부에서 동적으로 할당된 메모리의 주소를 반환 → OK**

```
1  int* CreateArray(int size, int initValue = 0) {
2     int* newStorage = nullptr;
3     newStorage = new int[size];
4     for (int i=0; i < size; ++i)
5        *(newStorage + i) = initValue;
6     return newStorage;
7  }
8
9  int main() {
10     int* myArray = nullptr;
11
12     myArray = CreateArray(100, 10);
13     delete[] myArray;
14     return 0;
15 }
```

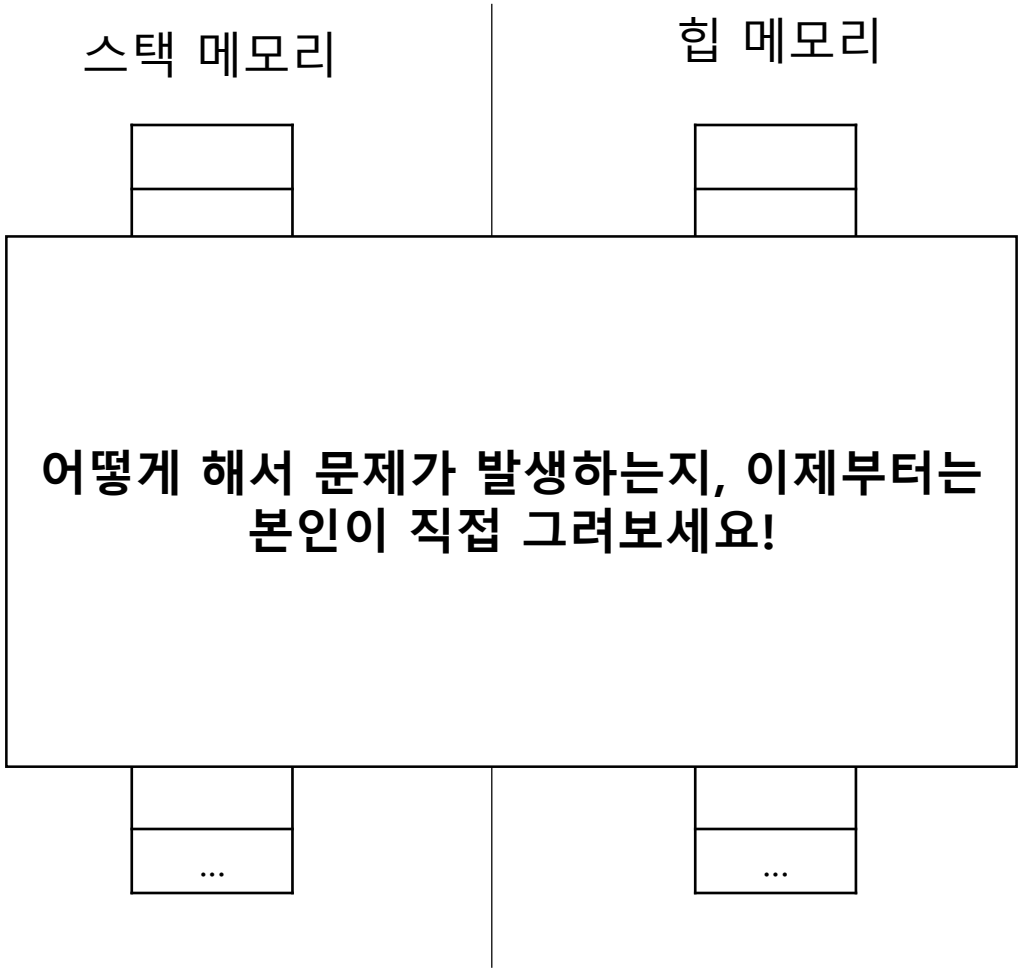

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Returning a Pointer

중요!

- 포인터의 반환
  - (주의!) 지역 변수에 대한 포인터 반환 → 안됨!!

```
1  int* DontDoThis()
2  {
3      int num = 10;
4      int* numPtr = &num;
5
6      return numPtr;
7  }
8
9  void main()
10 {
11     int* a = nullptr;
12     a = DontDoThis();
13     cout << *a << endl;
14 }
```



Pointer Cautions

- 초기화의 필요성

```
1 int* intPtr; // anywhere
2 ...
3 *intPtr = 100; // in VS, compiler protects
```

- 허상 포인터(dangling pointer)
  - 두 포인터가 동일 데이터를 가리키다, 하나의 포인터가 메모리를 해제할 경우
  - 지역 변수를 참조하고, 호출 스택이 끝나는 경우
- new의 실패
  - 가끔 발생할 수 있음. 이런 경우 예외 처리 필요(후반 강의)
- 메모리 누수(memory leak)
  - 동적 할당으로 사용한 힙 메모리는 반드시 해제해야 함!

(Summary) Pointers

- 포인터의 선언 (포인터는 변수, * 사용, nullptr 초기화)
- 주소로의 접근 (&, 주소의 크기와 데이터의 크기 구분)
- 역참조 (*, 포인터가 가리키는 데이터 접근)
- 동적 메모리 할당 (new, delete, 런타임에 힙 메모리 사용)
- 포인터와 배열 (주소값의 증감 offset)
- 포인터와 const (3가지 case, 주소/값의 const)
- 포인터의 pass-by-address(전달된 포인터, 내부에서 동적 할당한 포인터, 지역변수 반환 금지)
- 주의사항(초기화, 허상, new 실패, 메모리 누수)

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

참조자

(Summary) References

- 참조자란?
- l-value와 r-value
- 포인터 vs 참조자

Reference

중요!

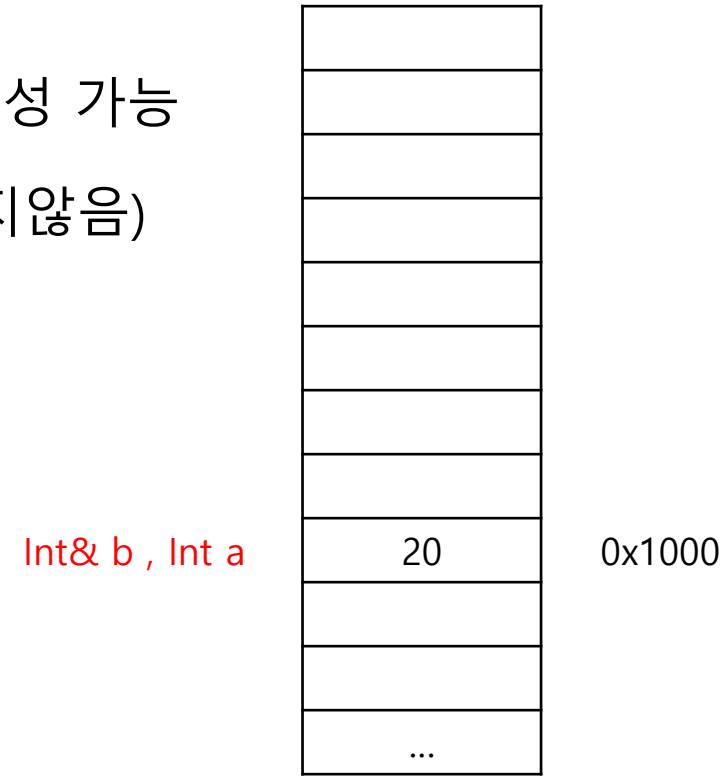
- 참조자
  - 변수의 별명
  - 참조자는 (새로운) 변수가 아님
    - 변수가 아니라는 의미는, 메모리에 새로운 공간을 차지하지 않는다는 의미
    - 포인터 변수는 그 자체가 주소값을 저장하는 하나의 변수였지만, 참조자는 다름
  - 선언과 동시에 초기화 되어야 함 (Null일 수 없음)
  - 한 번 초기화되면, 다른 변수의 참조자가 될 수 없음
  - Const인 pointer이면서, 사용 시 자동으로 역참조를 수행하는 개념
    - 포인터의 간단하고 편리한 버전으로 생각
  - C++에서, 함수의 매개변수로 "매우" 자주 사용

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Reference

- 참조자
  - int&, float& 등 변수 정의시에 &를 붙인 타입을 사용
  - 포인터와 마찬가지로, 동일한 타입에 대해서만 참조자 생성 가능
  - 참조자를 사용할 때는 마치 a인것처럼 그냥 사용(*,& 붙지않음)

```
1  int main()
2  {
3      int a = 10;
4      int& b = a;
5
6      b = 20; // 사용할 때는 그냥 사용!
7      cout << a << endl; // 20
8
9      int& c; // ERROR! 초기화가 반드시 필요
10             // 무엇의 별명인지 만들때 명시해야 함
11 }
```

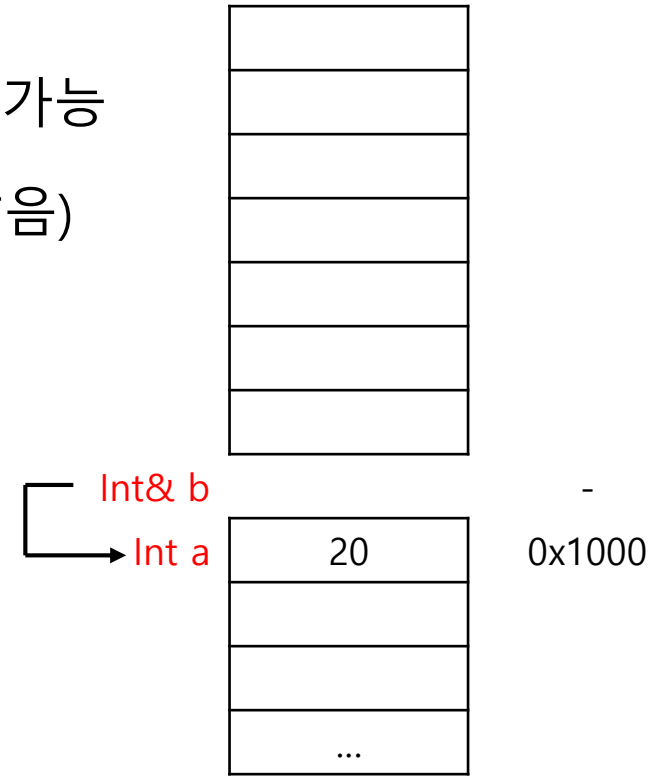


참조자인 b는 마치 a변수인 것처럼 사용 가능

Reference

- 참조자
  - int&, float& 등 변수 정의시에 &를 붙인 타입을 사용
  - 포인터와 마찬가지로, 동일한 타입에 대해서만 참조자 생성 가능
  - 참조자를 사용할 때는 마치 a인것처럼 그냥 사용(*,& 붙지않음)

```
1  int main()
2  {
3      int a = 10;
4      int& b = a;
5
6      b = 20; // 사용할 때는 그냥 사용!
7      cout << a << endl; // 20
8
9      int& c; // ERROR! 초기화가 반드시 필요
10             // 무엇의 별명인지 만들때 명시해야 함
11 }
```



참조자인 b는 마치 a인 것처럼 사용 가능.
그림으로는 위와 같이 표현하겠습니다.
(b는 주소값이 따로 없는 것에 유의)

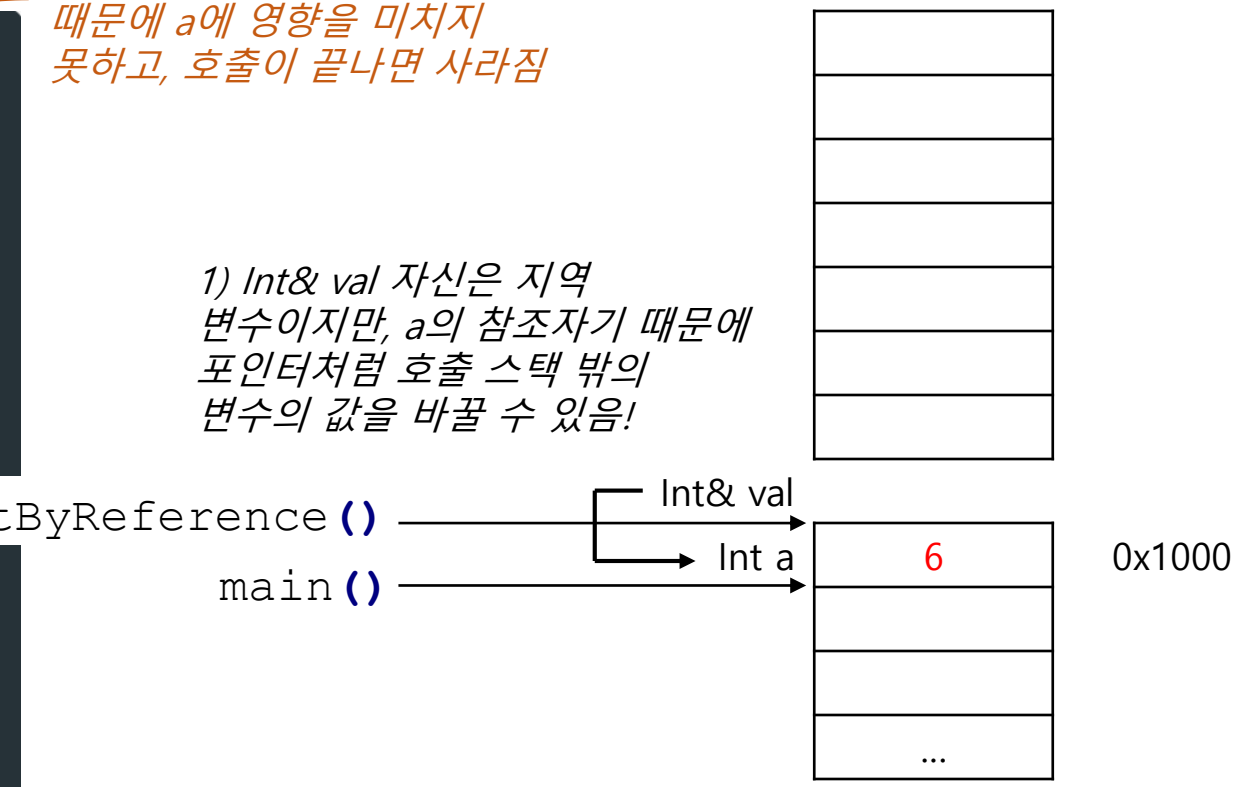
Reference

- 참조자

```
1  void Increment(int val)
2  {
3      val++;
4  }
5
6  void IncrementByReference(int& val)
7  {
8      val++;
9  }
10
11 int main()
12 {
13     int a = 5;
14     Increment(a);
15     cout << a << endl; //5
16
17     IncrementByReference(a);
18     cout << a << endl; //6
19
20     return 0;
21 }
```

2) 반면에 이 val은 단지 a의 값을 복사해온 지역변수이기 때문에 a에 영향을 미치지 못하고, 호출이 끝나면 사라짐

1) Int& val 자신은 지역 변수이지만, a의 참조자기 때문에 포인터처럼 호출 스택 밖의 변수의 값을 바꿀 수 있음!



문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Reference

- 참조자
  - Copy가 필요한 경우가 아니라면, const &를 쓰는 것이 기본
    - ➢ Const & rvalue 전달 가능
  - 오른쪽 코드는 모두 a의 값인 5를 그대로 출력함
    - ➢ 네개의 Print 함수의 장/단점은?

```
1  void PrintConstRef(const int& val)
2  {
3      cout << val << endl;
4  }
5  void PrintAddress(int* valPtr)
6  {
7      cout << *valPtr << endl;
8  }
9  void PrintRef(int& val)
10 {
11     cout << val << endl;
12 }
13 void PrintVal(int val)
14 {
15     cout << val << endl;
16 }
17 int main()
18 {
19     int a = 5;
20     PrintVal(a);
21     PrintRef(a);
22     PrintConstRef(a);
23     PrintAddress(&a);
24 }
```

Reference

고급

- 참조자가 새로운 변수가 아니라는 뜻의 의미
  - 메모리 디버깅를 통해 a의 주소와 b의 주소를 살펴보면 동일한 것을 알 수 있음
  - 즉, b가 (포인터와는 달리) 메모리 공간을 차지하는 새로운 변수는 아니라는 의미
  - (컴파일 시에 참조자는 가리키는 원본의 메모리 주소로 대체되는 방식)

```
1  int main()
2  {
3      int a = 2;
4      int& b = a;
5      int* c = &b;
6
7      cout << (c == &a) << endl;
8
9      return 0;
10 }
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

l-values and r-values

- l-value
  - 이름을 가지며, 주소를 갖는 값
  - Const가 아니라면 수정이 가능한 값

```
1 int x =100; //x is l-value
2
3 string name = "Hyungki"; //name is l-value
```

- r-value
  - 주소를 갖지 않고, 대입의 대상이 될 수 없음
  - L-value가 아닌 것들
    - ➢ 대입식의 오른쪽에 위치
    - ➢ 리터럴 등

```
1 int x = 100; //100 is r-value
2
3 string name = "Hyungki"; //"Hyungki" is r-value
4
5 int maxNum = max(20,30); //max(20,30) is r-value
```

l-values and r-values

고급

- 기본적으로는 L-value인 경우에만 참조자를 만들 수 있으나...
  - Const reference인 경우와
  - R-value reference에 대한 문법 또한 존재

```
1 int x = 100;
2
3 int& ref1 = x;
4 ref1 = 200;
5
6 int& ref2 = 100; //ERROR! 100 is not l-value
7 const int& ref3 = 100; //allowed
8 int&& ref4 = 100; //allowed, r-value reference
```

Pointers vs References

중요!

- 간단 정리
  - *의 사용
    - 변수를 정의할 때 붙는다? → 포인터의 정의(포인터 변수의 생성)
    - 변수를 사용할 때 붙는다? → 포인터의 역참조
  - &의 사용
    - 변수를 정의할 때 붙는다? → 참조자의 정의(참조자의 생성)
    - 변수를 사용할 때 붙는다? → 변수의 주소값 반환

```
int* a

*a

int& b

&a
```

Pointers vs References

- Pass-by-value `void func(int a)`
  - 함수가 실제 매개변수(원본)를 수정하지 않는 경우 사용
  - Int, char, double과 같은 크기가 작은 기본 자료형의 전달
    - 기본 자료형이 아닌 자료형? → string, vector, class
    - 복사의 비용이 높기 때문

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pointers vs References

- Pass-by-address(포인터 사용)

```
void func(int* a)
```

  - 함수가 실제 매개변수(원본)를 수정해야 하는 경우
    - 즉, 지역 범위 밖의 메모리에 저장된 값에 접근해야 하는 경우
  - 복사의 오버헤드가 클때
  - 포인터가 nullptr이 되어도 상관 없을 때
    - 참조자는 null이 될 수 없음

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pointers vs References

- const의 포인터를 사용한 Pass-by-address

```
void func(const int* a)
```

  - 함수가 실제 매개변수를 **수정하지 않는** 경우
  - 복사의 오버헤드가 클때
  - 포인터가 nullptr이 되어도 상관 없을 때
    - 참조자는 null이 될 수 없음

Pointers vs References

- const의 const인 포인터를 사용한 Pass-by-address `void func(const int* const a)`
  - 함수가 실제 매개변수를 **수정하지 않는** 경우
  - 복사의 오버헤드가 클때
  - 포인터가 nullptr이 되어도 상관 없을 때
    - ➢ 참조자는 null이 될 수 없음
  - **포인터가 바뀌지 않아야 할 때**

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

Pointers vs References

- Pass-by-reference(참조자 사용)

```
void func(int& a)
```

  - 함수가 실제 매개변수를 **수정하는** 경우
  - 복사의 오버헤드가 클 때
  - 매개변수가 **null이 되지 않는 것이 보장될 때**

Pointers vs References

- const 참조자를 사용한 Pass-by-const-reference

```
void func(const int& a)
```

  - 함수가 실제 매개변수를 **수정하지 않는** 경우
  - 복사의 오버헤드가 클 때
  - 매개변수가 **null이 되지 않는 것이 보장될 때**

(Summary) References

- 복습 (&, 변수의 별명, pass-by-reference)
- l-value와 r-value (주소값을 가질 수 있는 변수(l-value)가 참조자 생성 가능)
- 포인터 vs 참조자
  - Pass-by-value
  - Pass-by-reference
    - Pointer
    - Pointer to const
    - Const pointer to const
    - Reference
    - Const reference

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

추가 슬라이드

Pointer Arithmetic

- 포인터 연산
  - **대입 표현식**
  - **연산 표현식**
  - 비교 표현식

```
int_ptr++; // next array element
int_ptr--; // previous array element

int_ptr += n; // increment n*sizeof(type)
int_ptr -= n; // decrement n*sizeof(type)

int n = int_ptr2 - int_ptr1; // number of element between two pointers
```

Pointer Arithmetic

- 포인터 연산
  - 대입 표현식
  - 연산 표현식
  - **비교 표현식**
    - 값이 아닌 주소값의 비교 (값을 비교하고 싶다면 역참조 후 비교)

```
string s1="Frank";
string s2="Hyungki";

string* p1 =&s1;
string* p2 =&s2;
string* p3 =&s1;

cout << (p1==p2) << endl; // false
cout << (p1==p3) << endl; //true
```

```
string s1="Hyungki";
string s2="Hyungki";

string* p1 =&s1;
string* p2 =&s2;
string* p3 =&s1;

cout << (*p1==*p2) << endl; // true
cout << (*p1==*p3) << endl; // true
```