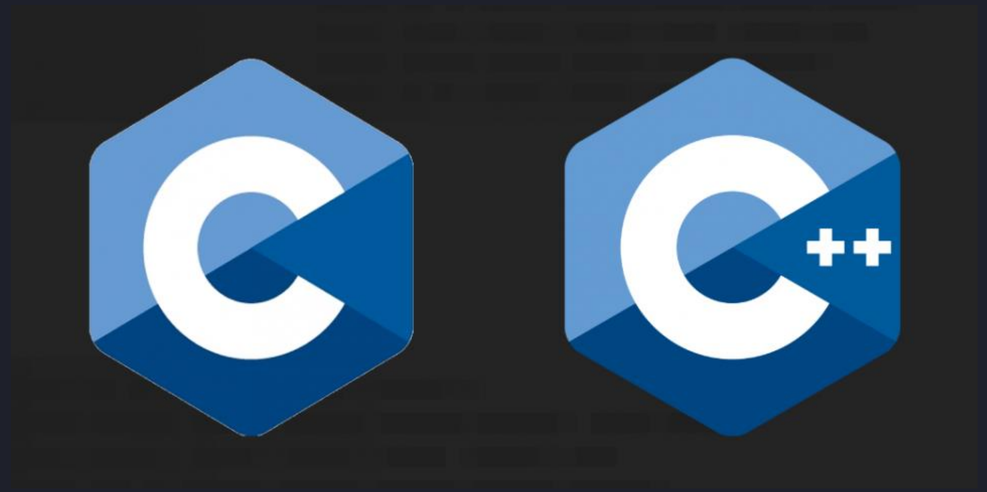


SPLIT 튜터링 6회차
:메모리 주소와 포인터

기계항공공학부 17학번 김기성

Contents

1 주소와 메모리

2 포인터

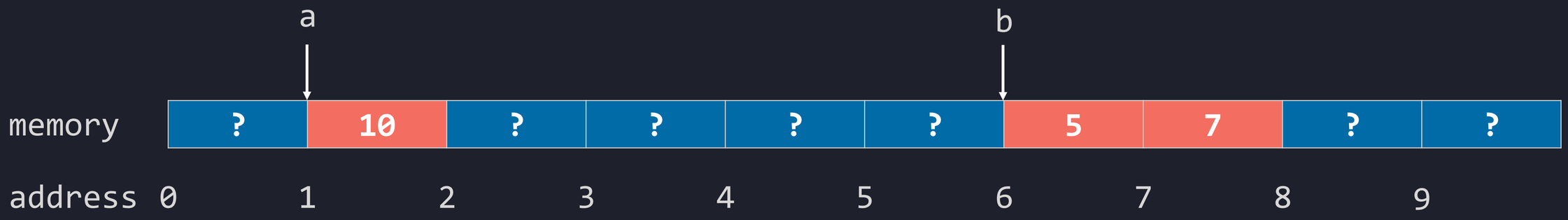
3 연습문제

주소(address)와 메모리(memory)

- 모든 변수들은 자신만의 주소가 있습니다.
  - 단일 변수일 경우, 해당 변수의 주소이고, array일 경우 가장 첫 원소의 주소입니다.
- 메모리를 땅 경매장에 비유해보겠습니다.
- 만약 길이 3짜리의 int array를 선언할 경우, 경매장에서 3블록 이상 크기의 매물의 주소를 찾아, 3블록을 해당 array에 할당합니다. 자투리 블록은 다시 경매장에 나갑니다. 변수가 선언된 코드 블록(=중괄호)이 끝나면, 그 블록에서 선언된 변수들은 수명이 끝납니다.
  - 해당 변수들이 차지하고 있던 메모리는 다시 경매장으로 나갑니다.

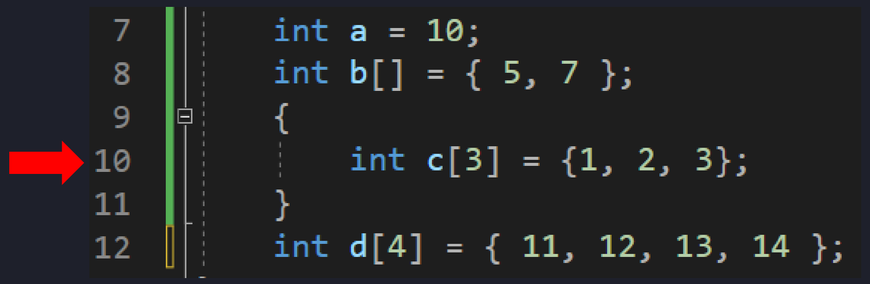
주소(address)와 메모리(memory)

- 예를 들어 메모리의 크기를 10이라고 생각해보겠습니다.(실제로는 훨씬 큽니다)
- 아래 코드에서 line 10 이전에는 크기 1짜리 변수(a)와, 크기 2짜리 array(b)가 있습니다.
- 경매장에는 3개의 블록이 매물로 나와있습니다.
- c를 할당할 알맞은 블록은 주소 2에서 시작하는 크기 4짜리 블록입니다.



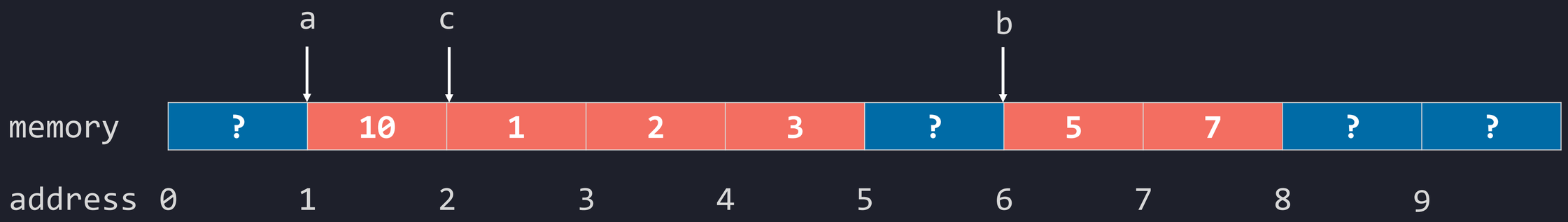
경매장

시작 주소	size
0	1
2	4
8	2



주소(address)와 메모리(memory)

- line11에서는 array c의 할당을 마친 상태입니다.
- 경매장에는 길이4의 매물이 사라지고, 자투리인 길이1의 매물이 새로 생겨났습니다.



경매장

시작 주소	size
0	1
5	1
8	2

```
7     int a = 10;
8     int b[] = { 5, 7 };
9     {
10        int c[3] = {1, 2, 3};
11    }
12    int d[4] = { 11, 12, 13, 14 };
```

주소(address)와 메모리(memory)

- line12에서는 array c의 수명이 끝났습니다.
- c가 차지하고 있던 블록은 주변의 블록과 합쳐져 경매장에 매물로 나옵니다.
- 경매장에는 길이4의 매물이 사라지고, 자투리인 길이1의 매물이 새로 생겨났습니다.
- c가 차지하고 있던 공간에 있던 값을 굳이 지우지는 않습니다(시간 절약을 위해)

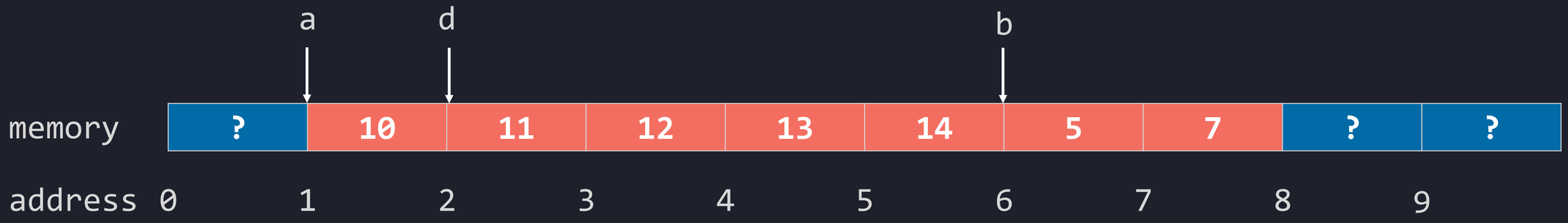
경매장

시작 주소	size
0	1
2	4
8	2

```
 7      int a = 10;
 8      int b[] = { 5, 7 };
 9      {
10          int c[3] = {1, 2, 3};
11      }
12      int d[4] = { 11, 12, 13, 14 };
```

주소(address)와 메모리(memory)

- line13에서는 array d를 할당합니다.
- 길이 4짜리 매물에 d를 할당합니다.
- 경매장에서는 길이4의 매물이 사라집니다.
- c가 있었던 공간에 있었던 값들은 d로 덮어씌워집니다.



경매장

시작 주소	size
0	1
8	2

```
7     int a = 10;
8     int b[] = { 5, 7 };
9     {
10        int c[3] = {1, 2, 3};
11    }
12    int d[4] = { 11, 12, 13, 14 };
```

변수의 주소 확인 방법

- &연산자(주소 연산자)를 이용하면 해당 변수의 주소를 확인할 수 있습니다.
- 변수의 주소는 보통 16진법으로 표시합니다.
  - 16진법이란? 0, 1, 2, 3, … 9, A, B, C, D, E, F의 16가지 심볼로 한 자리를 나타내는 방법
  - 주소는 000…000부터, FFF….FFF번지의 범위까지 가능합니다.

```
int main() {
    int a = 1;
    int b = 2;
    cout<<"a의 값: "<< a << endl;
    cout << "a의 주소: " << &a << endl;
    cout << "b의 값: " << b << endl;
    cout << "b의 주소: " << &b << endl;
    return 0;
}
```

```
a의 값: 1
a의 주소: 00000069B537FB74
b의 값: 2
b의 주소: 00000069B537FB94
```

포인터(pointer)

- 포인터란?
  - 포인터는 주소를 담을 수 있는 변수입니다.
- 데이터타입 뒤에 *을 붙이면 해당 타입의 포인터가 생성됩니다.
  - int* 타입 포인터는 int 변수의 주소를, double*타입의 포인터는 double 변수의 주소를 담을 수 있습니다.
- 포인터 변수에 *연산자(역참조 연산자)를 사용하면 해당 주소의 값을 읽거나 변경할 수 있습니다.

```cpp
int main() {
    int a = 10;
    int* p = &a; //p라는 포인터에 a의 주소를 저장
    cout << "a의 주소: " << &a << endl;
    cout << "p가 저장하고 있는 주소: " << p << endl;
    cout << "*p = " << *p << endl;
    cout << "p에 있는 값을 5로 변경합니다.\n";
    *p = 5;
    cout << "a = " << a << endl;
    return 0;
}
```

```
a의 주소: 0000000CBB1EF644
p가 저장하고 있는 주소: 0000000CBB1EF644
*p = 10
p에 있는 값을 5로 변경합니다.
a = 5
```

포인터(pointer)

- 포인터의 덧셈, 뺄셈
  - 포인터에 1을 더하거나 빼는 것은 해당 자료형의 크기(byte단위)만큼 포인터를 증가시키는 것입니다.
  - int는 4바이트이기 때문에 int*형 포인터에 1을 증가시키면 4가 증가합니다.

```
int a = 1;
int* p = &a;
cout << "p = " << p << endl;
cout << "p+1 = " << p + 1 << endl;
```

```
p = 0000006613F1FCC4
p+1 = 0000006613F1FCC8
```

  - long long 자료형은 8바이트이기 때문에 long long*형 포인터에 1을 증가시키면 8이 증가합니다.

```
long long a = 1;
long long* p = &a;
cout << "p = " << p << endl;
cout << "p+1 = " << p + 1 << endl;
```

```
p = 0000006FB2CFF818
p+1 = 0000006FB2CFF820
```

    - 16진법으로 표시되어있기 때문에 18->19->1A->1B->1C->1D->1E->1F->20 순서로 증가합니다.

포인터를 사용하는 이유

- 함수에 매개변수로 전달할 시, 해당 변수의 원본을 조작할 수 있습니다.
  - '=' 연산자는 변수의 복사본을 다른 변수에 할당합니다.
  - 포인터를 전달하면 해당 변수의 주소에 접근할 수 있게 됩니다.
- 변수 복사에 필요한 비용을 줄일 수 있습니다.
  - int같은 기본 자료형은 크기가 작아서 괜찮지만, string, vector같이 자료형은 크기가 클 수록 복사에 걸리는 시간이 길어집니다.

```cpp
#include <iostream>
#include <string>
#include <time.h>

using namespace std;

int main() {
	string a(100000, 'a'); // a = "a...a"(length = 100000)
	clock_t start, end;
	start = clock();
	string b = a; // 100000 times of copy occurs
	end = clock();
	auto duration = (double)(end - start)/CLOCKS_PER_SEC;
	cout << "수행 시간 : " << duration << "초"<<endl;
	return 0;
}
```

```cpp
#include <iostream>

using namespace std;
void f(int k) {
	k = 1;
	return;
}

void g(int* k) {
	*k = 1;
	return;
}
int main() {
	int a = 0;
	f(a);
	cout << a << endl;
	g(&a);
	cout << a << endl;
	return 0;
}
```

배열과 포인터

- 배열과 포인터는 매우 유사합니다.
- 배열을 선언할 때, 배열의 이름은 해당 배열의 첫번째 원소의 주소를 저장하고 있습니다.

```
int a[] = { 1, 2, 3 };
cout << "a = "<<a << endl;
cout << "a[0]의 주소 = " << &(a[0]) << endl;
```

```
a = 000000B1424FFA58
a[0]의 주소 = 000000B1424FFA58
```

- 배열에 '[]'연산자를 사용하는 것은 다음과 같은 형태라는 것을 알 수 있습니다.
  - (첫번째 원소의 주소)[인덱스] = 첫번째 주소로 부터 '인덱스'만큼 떨어진 곳에 있는 원소
- C++에서는 포인터를 배열처럼 사용할 수 있도록 되어있습니다.
  - p[idx] = p 주소로부터 idx만큼 떨어진 주소에 저장된 내용

```
int a[] = { 1, 2, 3 };
int* p = a; // p = a의 첫번째 원소의 주소
cout << p[1]; // 2
```

nullptr(널포인터)

- 포인터는 선언할 때 그 값을 꼭 지정해주는 것이 좋습니다.
  - 지정해주지 않았을 경우, 아무 값이 대입되어 있고, 어떤 주소를 가리키게 될지 모릅니다.
- 아무것도 가리키지 않는 포인터는 nullptr을 대입하여 초기화하는 것이 좋습니다.
  - nullptr은 해당 포인터 변수가 아무것도 가리키지 않음을 명시적으로 표현합니다.
  - 따라서 프로그래머가 해당 포인터가 nullptr인지 체크할 수 있습니다.

```
int main() {
    int* p1; // error!
    int* p2 = nullptr;

    return 0;
}
```

->연산자

- class를 가리키는 포인터의 멤버함수를 사용하는 방법은 다음과 같습니다.
  - 방법1: '*'연산자를 이용해 해당 주소를 역참조한 후 '.'연산자 사용

```
string s = "Hello, World!";
string* p = &s;
cout << (*p).length() << endl;
```

  - 방법 2: 포인터에 ->연산자를 사용하기

```
string s = "Hello, World!";
string* p = &s;
cout << p->length() << endl;
```

  - 방법 2를 사용할 수 있는 상황에는 방법 2를 사용하는 것이 가독서 면에서 더 낫습니다.
  - 그러나 멤버 연산자(+, [] 등)을 사용할 때에는 역참조를 사용해야 합니다.
    - ex) 포인터를 이용해 s의 세번째 원소에 접근하기

```
string s = "Hello, World!";
string* p = &s;
cout << (*p)[3] << endl;
```

연습문제

- 연습문제 1) 정수를 매개변수로 입력받아 1만큼 증가시키는 함수를 구현하고 싶습니다.
  - 아래 함수를 구현하고 그 결과를 확인해봅시다.
  - 의도한대로 작동이 안되는 이유는 무엇일까요?

```
void increase_by_one(int x) {
    x += 1;
    return;
}

int main() {
    int a = 0;
    increase_by_one(a);
    cout << a << endl;
    return 0;
}
```

- 연습문제 2) 정수의 포인터를 매개변수로 입력받아 해당 주소에 있는 값을 1 증가시키는 함수를 구현해봅시다.