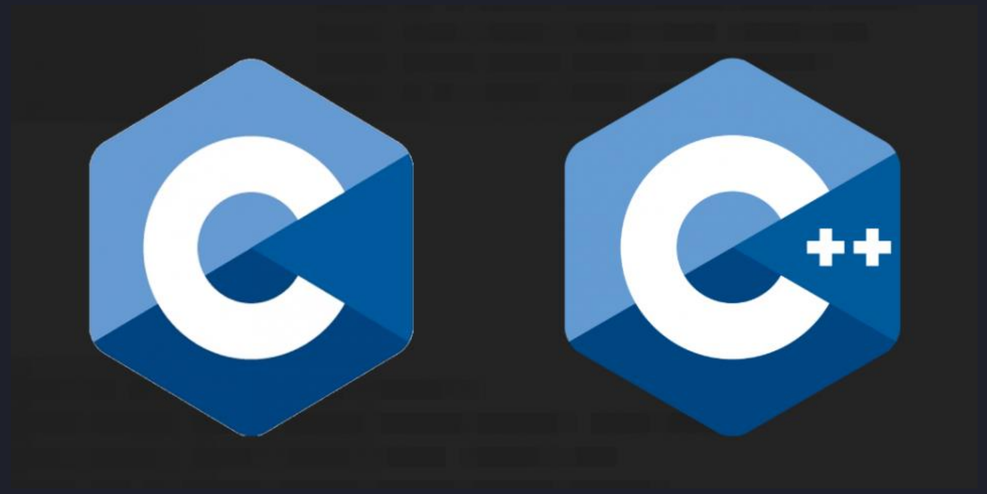


SPLIT 튜터링 7회차

:참조에 의한 전달, 동적 할당

기계항공공학부 17학번 김기성

Contents

1 pass by reference

2 동적 할당

3 연습문제

Pass by reference

- C++에서 할당 연산자 '='는 원본이 아니라 복사본을 생성해 새로운 변수에 할당합니다(pass by value).

```
int a = 1;
int b = a;
b = 2;
cout << "a는 "<<a <<"입니다."<< endl;
```

```
a는 1입니다.
```

- 이는 함수에 매개변수로 전달할 때에도 마찬가지입니다.

```
void f(int x) {
    x = 2;
    return;
}
int main() {
    int a = 1;
    f(a); // x = a;(복사본 생성)
    cout << "a는 "<<a <<"입니다."<< endl;
    return 0;
}
```

```
a는 1입니다.
```

```
void f(int *x) {
    (*x) = 2;
    return;
}
int main() {
    int a = 1;
    f(&a);
    cout << "a는 "<<a <<"입니다."<< endl;
    return 0;
}
```

Pass by reference

- 만약 함수 내부에서 원본을 조작하고 싶다면, int가 아니라 해당 변수의 주소를 전달하는 방법이 있습니다.

```
void f(int *x) {
    (*x) = 2;
    return;
}
int main() {
    int a = 1;
    f(&a);
    cout << "a는 "<<a <<"입니다."<< endl;
    return 0;
}
```

```
a는 2입니다.
```

- 포인터를 이용하지 않고, 원본을 전달하는 방법이 있을까요?
  - pass by reference(참조에 의한 전달)을 사용하면 됩니다.
    - 전달 받는 변수를 참조형 변수로 선언.(type 뒤에 &을 붙임)하고, 원본 변수로 초기화합니다.
    - b를 수정하면 원본을 수정하는 것과 같기 때문에 a도 함께 바뀝니다.

```
int a = 1;
int& b = a;
b = 2;
cout << "a는 "<< a <<"입니다."<< endl;
```

Pass by reference

- Pass by reference는 함수의 매개변수에도 적용할 수 있습니다.

```cpp
void f(int& x) {
    x = 2;
    return;
}
int main() {
    int a = 1;
    f(a); // a의 원본을 전달
    cout << "a는 "<< a <<"입니다."<< endl;
    return 0;
}
```

```
a는 2입니다.
```

- pass by reference를 사용하는 이유가 무엇일까요?
  - pointer로 전달하는 이유와 같습니다.
    - 1) 원본에 대한 조작을 가능하게 하고,
    - 2) 복사본 생성을 하지 않기 때문에 빠릅니다.
      - 긴 string이나 vector를 전달할 때 특히나 중요합니다.

Pass by reference 연습문제

- 연습문제 1) swap 함수 구현하기
  - 두 개의 int a, b를 입력받아 a와 b에 저장된 값을 바꾸는 swap 함수를 구현하세요.
  - 다음과 같이 동작해야 합니다.

```
int main() {
    int a = 1, b = 2;
    swap(a, b);
    cout << "a는 "<< a <<", b는 "<<b<<"입니다."<< endl;
    return 0;
}
```

```
a는 2, b는 1입니다.
```

- 연습문제 2) pass by reference의 속도 체감하기
  - 오른쪽 코드를 실행해보고 실행시간 차이를 알아봅시다.
  - pass by value와 pass by reference를 각각 어떤 경우에 사용하는 것이 좋을까요?

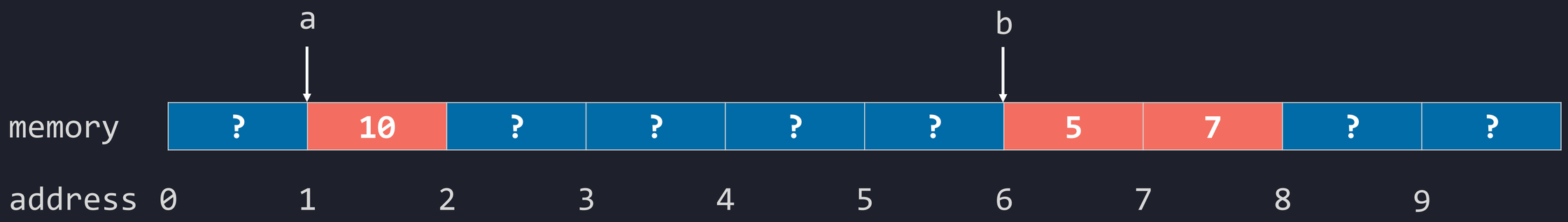
```
void f1(vector<int>& a) {
    cout << "f1 is called\n";
    return;
}

void f2(vector<int> a) {
    cout << "f2 is called\n";
    return;
}

int main() {
    vector<int> v(10000000, 1);
    for (int i = 0; i < 1000; ++i)
        f1(v);
    for (int i = 0; i < 1000; ++i)
        f2(v);
    return 0;
}
```

동적 할당

- 기존의 array 선언 방법
  - 기존에 변수나 array를 선언할 때에는 그 크기를 '프로그래밍'단계에서 정해야 했습니다.
  - 만약 프로그램이 돌아가다가 그보다 큰 크기로 커져야할 필요성이 생기면, 새로 프로그램을 짜야 했습니다.
  - 이를 크기가 정해져 있다고 해서 '정적할당'이라고 합니다.
  - 변수나 array가 차지하는 메모리 블록은, 해당 코드가 소속된 블록이 끝날 때 해제됩니다.
- '동적 할당'이란?
  - 프로그램이 실행되면서 메모리를 할당하고 해제하는 것을 의미합니다.
  - 메모리를 유연하게 확보하거나 해제할 수 있습니다.
    - 즉, 메모리를 효율적으로 관리할 수 있게 됩니다.
    - 다만 프로그래머 입장에서 신경써야 할 것들이 늘어납니다.
  - 해당 메모리 블록의 수명은 프로그래머가 정해줘야 합니다.
  - 그렇지 않으면 프로그램이 돌아가는 동안 해당 메모리 영역을 계속 잡아먹는 '메모리 누수'가 발생합니다.

정적 할당

- 경매장에는 3개의 블록이 매물로 나와있습니다.
- c를 할당할 알맞은 블록은 주소 2에서 시작하는 크기 4짜리 블록입니다.



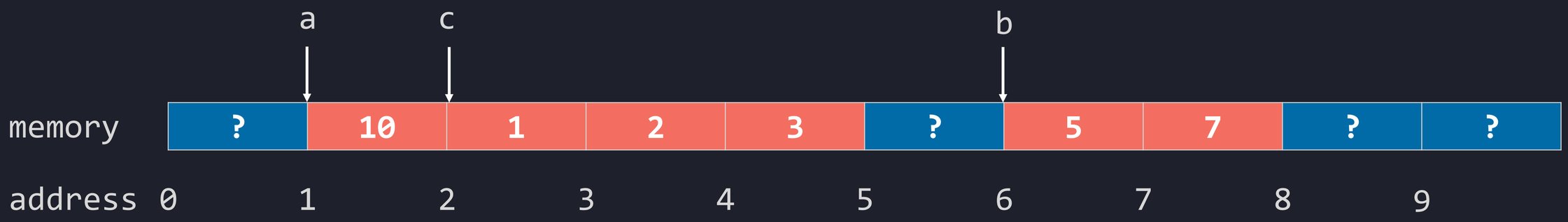
경매장

시작 주소	size
0	1
2	4
8	2

```
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 };
```

정적 할당

- 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 };
```

정적 할당

- 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 };
```

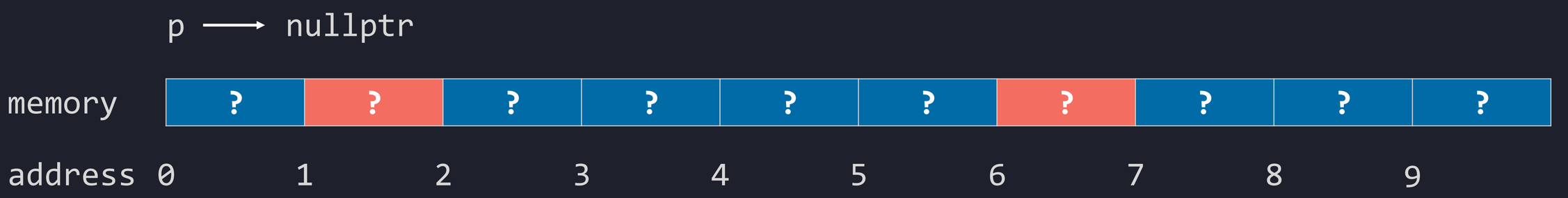
동적 할당

- new 키워드
  - 메모리 할당을 위한 키워드입니다.
  - new ‘type’[‘size’] 형태로 사용하면, 해당 타입의 변수 배열이 생성되고, 첫번째 원소의 주소를 반환합니다.
  - 배열이 아니라 한 개만 생성할 경우에는 new ‘type’형태로 사용하면 됩니다.
  - 반환된 주소는 배열처럼 사용할 수 있습니다.
- delete 키워드
  - 메모리 해제를 위한 키워드입니다.
  - 배열로 생성한 경우에는 delete[] 형태로 사용해야 합니다.

```
int* a = new int; // 정수 1개 동적 할당하는 법
int* b = new int[3]; // b: 길이 3의 배열
cout << a << endl; // a가 가리키는 주소
cout << b << endl; // b가 가리키는 주소(==b[0]의 주소)
delete a;
delete[] b;
```

동적 할당

- new 키워드는 경매장에서 땅을 구입하는 것, delete 또는 delete[]는 땅을 내놓는 것과 같습니다.
- new로 구입한 땅은 delete되기 전까지는 '절대' 매물로 나오지 않습니다.
  - 즉, 해당 메모리 공간의 수명을 프로그래머가 정하는 것이 됩니다.
  - new로 할당한 메모리를 적절한 시기에 delete해주지 않으면 쓸데없이 메모리를 차지하는 '메모리 누수'가 발생합니다.



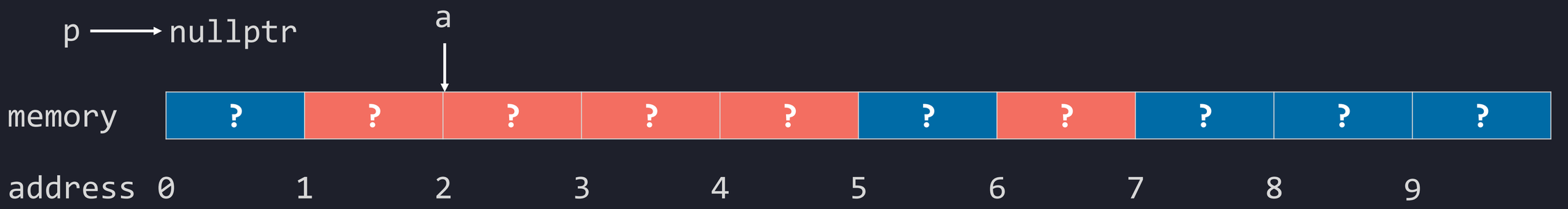
경매장

시작 주소	size
0	1
2	4
7	3

```
int* p = nullptr;
{
    int* a = new int[3];
    p = a;
    int b[] = { 1, 2, 3 };
}
delete[] p;
```

동적 할당

- a 포인터에 길이 3짜리 int 배열을 생성합니다. 경매장에서 적절한 크기의 공간을 찾아 할당합니다.
- 주소 2에 길이 3짜리 메모리가 할당되었습니다.



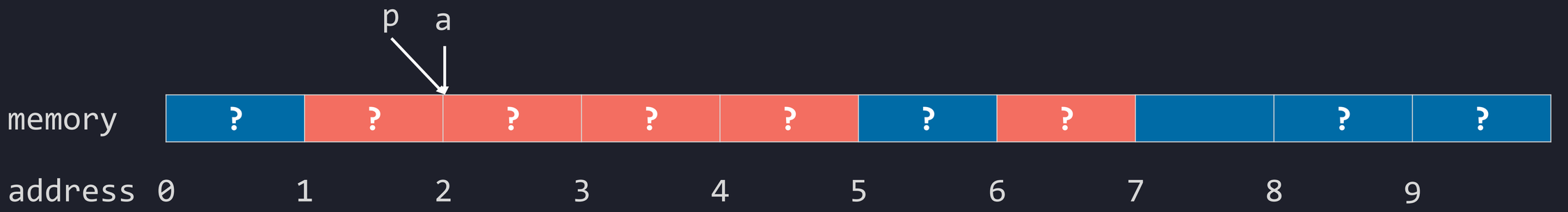
경매장

시작 주소	size
0	1
5	1
7	3

```
int* p = nullptr;
{
    int* a = new int[3];
    p = a;
    int b[] = { 1, 2, 3 };
}
delete[] p;
```

동적 할당

- p와 a가 같은 주소를 저장하게 됩니다.



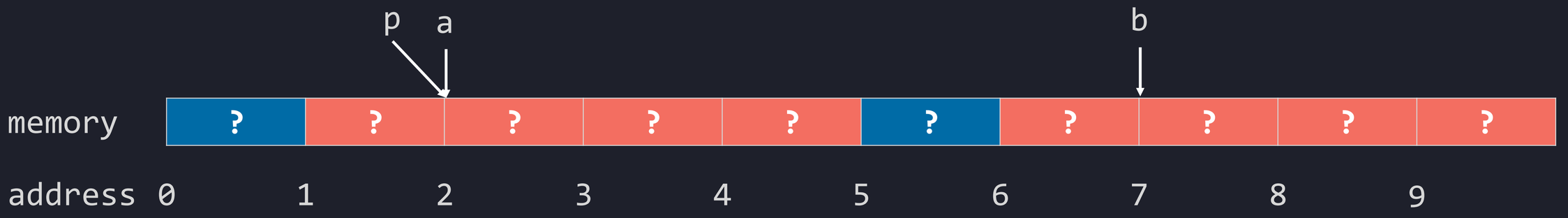
경매장

시작 주소	size
0	1
5	1
7	3

```
int* p = nullptr;
{
    int* a = new int[3];
    p = a;
    int b[] = { 1, 2, 3 };
}
delete[] p;
```

동적 할당

- 배열 b가 정적 할당됩니다.
- 주소 7에 길이 3짜리 공간이 할당됩니다.



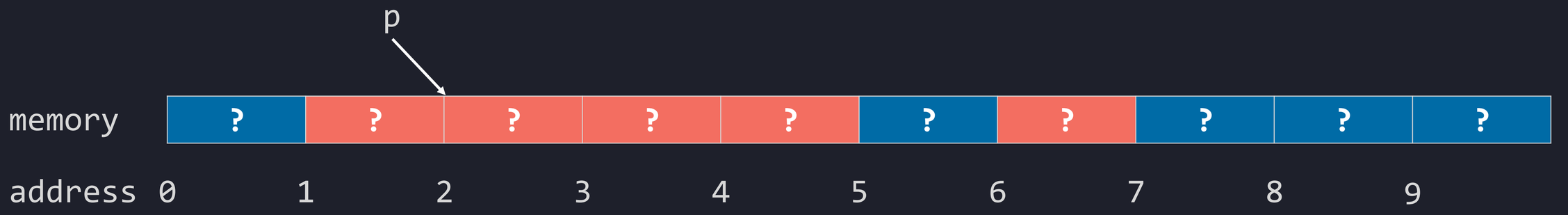
경매장

시작 주소	size
0	1
5	1

```
int* p = nullptr;
{
    int* a = new int[3];
    p = a;
    int b[] = { 1, 2, 3 };
}
delete[] p;
```

동적 할당

- 정적할당으로 생성된 배열 b가 생성된 블록(중괄호)이 끝났으므로 b가 차지하고 있던 메모리 공간은 자동으로 해제됩니다.
- 포인터 a도 해당 블록에서 생성된 지역변수이므로 사라지지만, a가 가리키고 있던 영역은 해제되지 않고 남아있습니다.



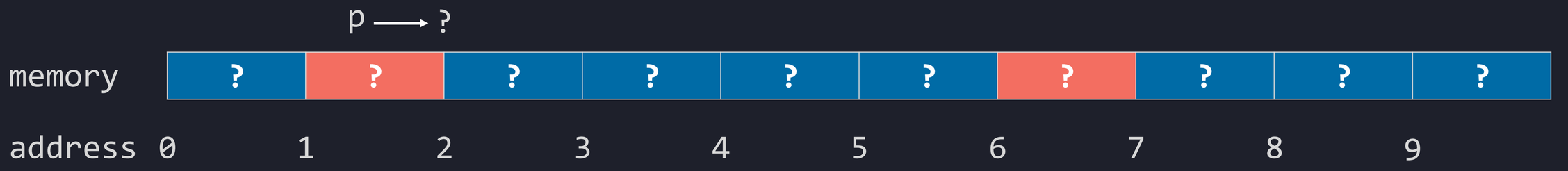
경매장

시작 주소	size
0	1
5	1
7	3

```
int* p = nullptr;
{
    int* a = new int[3];
    p = a;
    int b[] = { 1, 2, 3 };
}
delete[] p;
```

동적 할당

- p가 가리키고 있는 메모리 공간이 delete[] 키워드에 의해 해제됩니다.
- 주소 2에 있던 크기 3의 공간이 경매장으로 반환됩니다.
- delete는 해당 주소에 있는 메모리를 반환할 뿐, p가 저장하고 있던 주소는 바뀌지 않습니다.
  - 그러나 경매장에 나와있는 매물을 가리키고 있는 것은 보안상 위험하기 때문에 (use-after free 취약점), 컴파일러 버전에 따라 p가 쓰레기 값을 가리키게 하는 경우도 있습니다.



경매장

시작 주소	size
0	1
2	4
7	3

```
int* p = nullptr;
{
    int* a = new int[3];
    p = a;
    int b[] = { 1, 2, 3 };
}
delete[] p;
```

동적 할당 활용 예시

- 연습문제) 거꾸로 출력하기
  - 유저로부터 배열의 크기 N을 입력받으세요.
  - 이후 N개의 정수 입력을 입력하면, 해당 정수 수열을 거꾸로 출력하는 프로그램을 작성하세요.
  - 크기 N의 정수 배열을 동적 할당하여 구현하세요.
  - 프로그램 종료 이전에 해당 배열을 해제해주세요.

```
N을 입력하세요: 5
5개의 정수를 입력하세요
1 2 3 4 5
거꾸로 출력합니다.
5 4 3 2 1
```

```cpp
int main() {
    int N;
    cout << "N을 입력하세요: ";
    cin >> N;
    int* a = new int[N];
    cout << N << "개의 정수를 입력하세요\n";
    for (int i = 0; i < N; ++i) {
        cin >> a[i];
    }
    cout << "거꾸로 출력합니다.\n";
    for (int i = N - 1; i >= 0; --i) {
        cout << a[i] << ' ';
    }
    delete[] a;
    return 0;
}
```

동적 할당 활용 예시

- 정적 할당의 경우 여유롭게 공간을 할당하였지만, N이 작을 경우 메모리가 낭비되는 단점이 있습니다.
  - 또한, N이 10만보다 클 경우에는 버그가 발생합니다.
- 동적 할당의 경우 유저의 입력에 알맞은 사이즈의 배열을 생성할 수 있어 메모리 공간을 효율적으로 사용할 수 있습니다.

```cpp
int main() {
    int N;
    cout << "N을 입력하세요: ";
    cin >> N;
    int a[100000];
    cout << N << "개의 정수를 입력하세요\n";
    for (int i = 0; i < N; ++i) {
        cin >> a[i];
    }
    cout << "거꾸로 출력합니다.\n";
    for (int i = N - 1; i >= 0; --i) {
        cout << a[i] << ' ';
    }
    return 0;
}
```

```cpp
int main() {
    int N;
    cout << "N을 입력하세요: ";
    cin >> N;
    int* a = new int[N];
    cout << N << "개의 정수를 입력하세요\n";
    for (int i = 0; i < N; ++i) {
        cin >> a[i];
    }
    cout << "거꾸로 출력합니다.\n";
    for (int i = N - 1; i >= 0; --i) {
        cout << a[i] << ' ';
    }
    delete[] a;
    return 0;
}
```

여담: 메모리의 생김새

- 앞의 경매장 예시를 보면, 메모리가 파편화되어 있는 것을 볼 수 있습니다.
  - 메모리가 파편화되어있을 경우 여러 면에서 시간적, 공간적 손해가 발생합니다.
    - 파편화된 메모리에서 알맞은 공간을 찾는 것
    - 메모리를 해제할 때 주변 블록들과 합쳐지는 것
  - 더 효율적으로 할 수는 없을까요?
  - 또한 지역변수도 언제 할당되고 해제될 지 명확합니다(선언된 블록 내부에서만 존재).
    - array의 크기는 미리 알 수 있습니다.(코드에 명시되어 있으므로)
  - 동적 할당되는 변수는 그 크기를 예측할 수 없습니다. -> 파편화를 피할 수 없습니다.
  - 따라서 동적할당되지 않은 지역변수는 차곡차곡 정리할 수 있습니다.

여담: 메모리의 생김새

- 메모리의 Stack영역과 Heap 영역
  - 이 문제를 해결하기 위해 메모리 중 일부를 차곡차곡 정리하는 공간(stack)으로 구분해두었습니다.
  - 지역변수는 stack 영역에 저장됩니다. 차곡차곡 쌓는 방식으로 메모리를 할당하고 해제합니다.
    - 파편화가 일어나지 않고 할당, 해제, 접근 모두 heap 영역에 비해 빠릅니다.
    - 전체 메모리중 아주 작은 부분(MB단위)만 사용할 수 있습니다. -> Stack Overflow
  - 동적할당되는 변수들은 막 쌓아두는 heap 영역에 저장됩니다. 그 크기를 예측할 수 없으므로, 앞의 경매장 예시와 같은 방식으로 할당, 해제됩니다.

Stack

Heap