Ch7. 메모리 관리

Jong-Hyeok Park
akindo19@gmail.com

동적 메모리 관리

- 자동 변수
  - scope 벗어나면, 할당된 메모리 자동 해제

동적 메모리 관리

- 동적 메모리 할당
  - 메모리 해제 필요

동적 메모리 관리

- 동적 메모리 할당
  - 메모리 해제 필요


```
int** handle = nullptr;
handle = new int*;
*handle = new int;
```

메모리 할당과 해제

- new & delete
  - malloc() & free() 는 ~~절대로~~ 되도록 사용하지 말자!

```
void leaky()
{
    new int;
}
```

```
void safe()
{
    int* ptr = new int;
    delete ptr;
    ptr = nullptr;
}
```

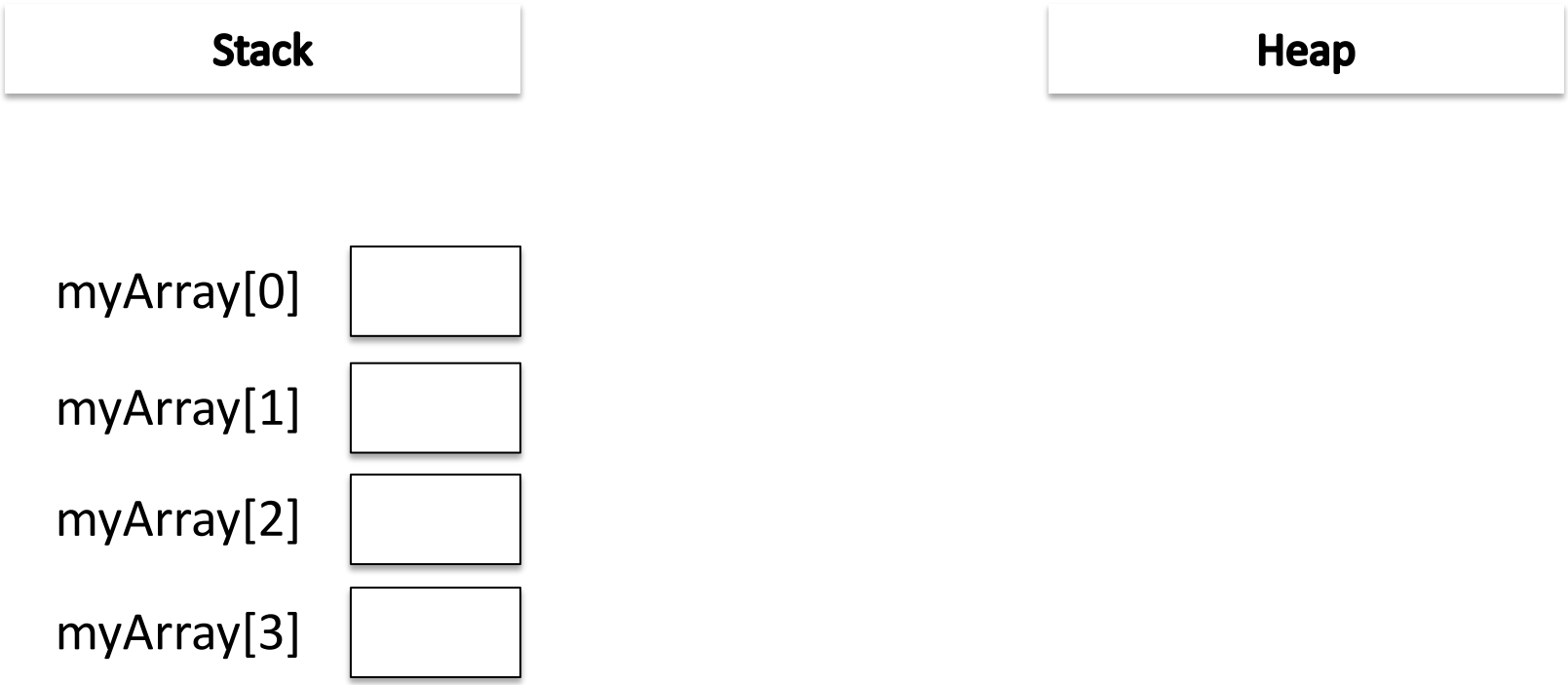
메모리 할당과 해제

- 메모리 할당 실패한 경우
  - nothrow: exception 대신, nullptr 리턴

```
void safest()
{
    int* ptr = new(nothrow) int;
    delete ptr;
    ptr = nullptr;
}
```

배열

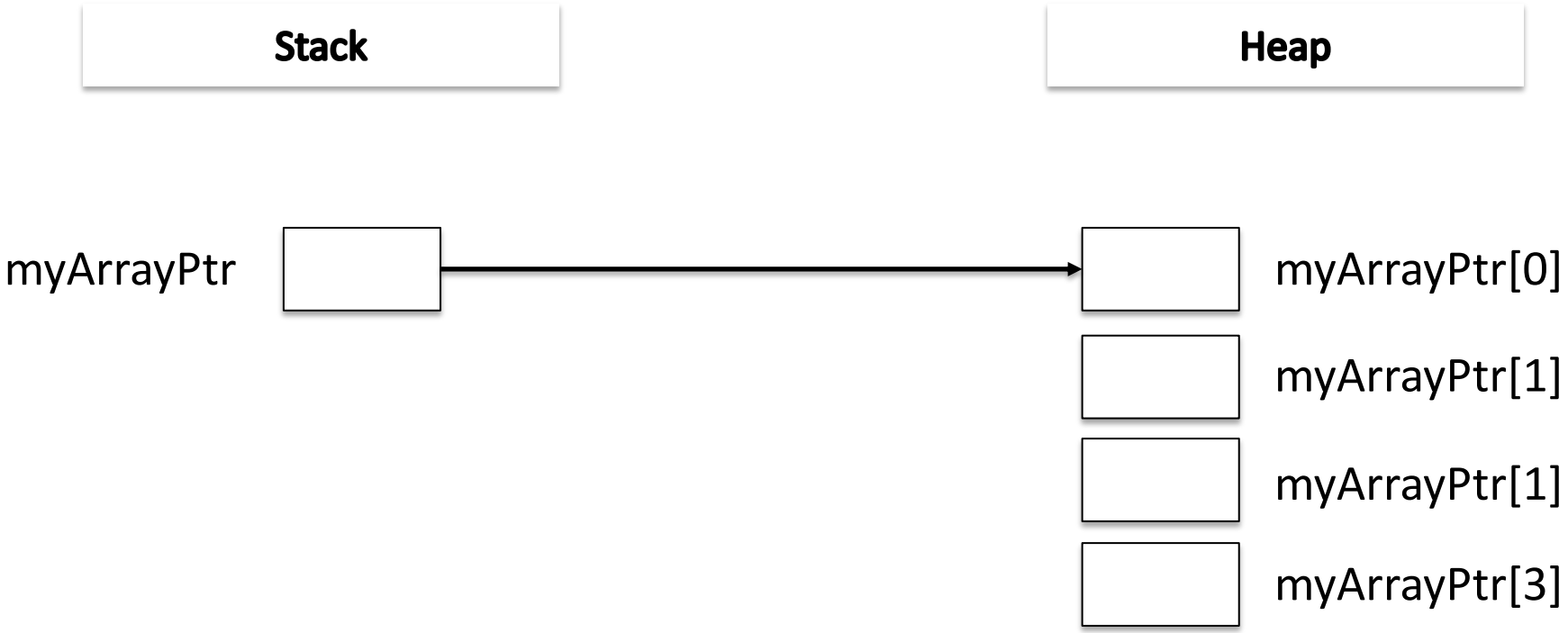
- 정적배열




```
int myArray[5];
```

배열

- 동적배열
  - 객체 배열도 가능




```
int* myArrayPtr = new int[5];
```

배열

- 다차원 배열
  - Heap에서는 메모리 공간이 연속적으로 할당되지 않기 때문에 주의 필요

```
char **board = new char [row][col];
// bug! compile error!
```

```
char** alloc(size_t row, size_t col)
{
    char** myArray = new char*[row];

    for (size_t i=0; i<row; ++i) {
        myArray[i] = new char[col];
    }

    return myArray;
}
```

배열

- 삭제
  - delete[]

```
Simple* sArr = new Simple[4];
delete[] sArr;
sArr = nullptr;
```

```
void release(char** arr, size_t row)
{
    for (size_t i=0; i<row; ++i){
        delete[] arr[i];
    }
    delete[] arr;
}
```

포인터

- 메모리 주소 (접근 주의 !)
- 포인터 작동 방식
  - * (역참조) : 값 얻기
  - & : 주소 얻기
- 타입 캐스팅
  - 안전하게 static_cast<> 사용!
  - **상속**관계 있는 대상끼리 캐스팅 할 경우 **동적 캐스팅** 사용

배열과 포인터

- 배열은 사실 첫번째 원소에 대한 **주소**이다.
- 함수 전달 : Stack 배열 vs. Heap 배열
  - Stack 배열: 컴파일러가 자동으로 포인터 변환
- 포인터는 모두 배열은 아니다!

```
int stackArr[] = {1, 3, 5, 7};
size_t size = std::size(stackArr);

stackFunc(stackArr, arrSize);
stackFunc(&stackArr[0], arrSize);
```

```
size_t size = 4;
int* heapArr = new
int[size]{1,3,5,7};

HeapFunc(heapArr, arrSize);
delete[] heapArr;
heapArr = nullptr;
```

배열과 포인터

- 포인터 연산
  - 포인터 뺄셈 : 바이트 (X) 지정한 타입 갯수 (O)

```
myArr[2] = 33;

*(myArr + 2) = 33;
```

GC

- C++ 은 GC 제공 안함.
- Mark & Sweep 기법
  - 모든 포인터를 Garbage Collector List에 등록
  - Garbage Collector에서 객체 상태 표시 가능하도록 *GarbageCollectable* 믹스인 클래스 상속하도록 함.
  - Garbage Collector 동작할 때 객체 접근하지 못하도록 함.
- 단점
  - Stall
  - 소멸자의 비결정적 호출

객체 풀

- 재활용
- 타입이 같은 여러 객체를 지속적으로 사용하는 경우
- E.g., Thread pool (멀티 쓰레드 활용!!!)

스마트 포인터

- 템플릿을 사용한 타입 세이프한 스마트 포인터 클래스
- 연산자 오버로딩
  - *, ->
  - 스마트 포인터 객체를 일반 포인터처럼 사용 가능
- 고유 소유권 방식
  - std:unique_ptr
- 공유 소유권 방식
  - std::shared_ptr

unique_ptr

- 동적 할당 리소스는 항상 unique_ptr 사용
- make_unique<> 사용

```
void leaky()
{
    Simpe* sPtr = new Simple();
    sPtr->go();
    // bug! no deallcoation
}

void Couldleaky()
{
    Simpe* sPtr = new Simple();
    sPtr->go();
    delete sPtr;
    // warning! go() exception
}
```

```
void safe()
{
    auto uPtr =
    make_unique<Simple>();

    uPtr->go();
}
```

unique_ptr

- get()
  - 내부 포인터 접근
- reset()
  - unique_ptr 내부 포인터 해제하고 다른 포인터로 변경 가능

```
uPtr.reset()                  // null 초기화
uPtr.reset(new Simple())  // 새로운 인스턴스 초기화
```

- release()
  - 내부 포인터 반환 후, 스마트 포인터를 nullptr로 설정
  - 그 후, 사용자가 직접 메모리 해제
- 복사 불가능
  - 이동 가능 ! std::move()

unique_ptr

- 배열
  - C 스타일 배열 저장 적합
  - std::array, std::vector 사용 추천

- 커스텀 제거자
  - unique_ptr 스코프 벗어날 때 리소스 자동으로 닫을 때 활용

```
int* malloc_int(int val)
{
    int* p =
    (int*)malloc(sizeof(int));
    *p = val;
    return p;
}
```

```
int main()
{
    unique_ptr<int, decltype(free)*>
    mySmartPtr(malloc_int(42), free);
    return0;
}
```

shared_ptr

- make_shared<> 사용
- get(), reset() 제공
  - reset() : 레퍼런스 카운팅 메커니즘에 따라, 마지막 shared_ptr제거/reset 될때만 리소스 해제함.
- release() 지원 X
- use_count()
  - 현재 동일한 리소스 공유하는 shared_ptr 갯수 반환

shared_ptr

- 커스텀 제거자
  - 커스텀 제거자 타입을 템플릿타입 매개변수 지정 불필요!

```
shared_ptr<int>
mySmartPtr(malloc_int(42, free);
```

- 캐스팅
  - const_pointer_cast() , dynamic_pointer_cast()
  - reinterpret_pointer_cast (c++17 이상)

shared_ptr

- 레퍼런스 카운팅
  - 현재 사용중인 객체 (스마트 포인터) 추적하는 기법
  - 스마트 포인터 중복 삭제 방지
  - unique_ptr은 지원 X

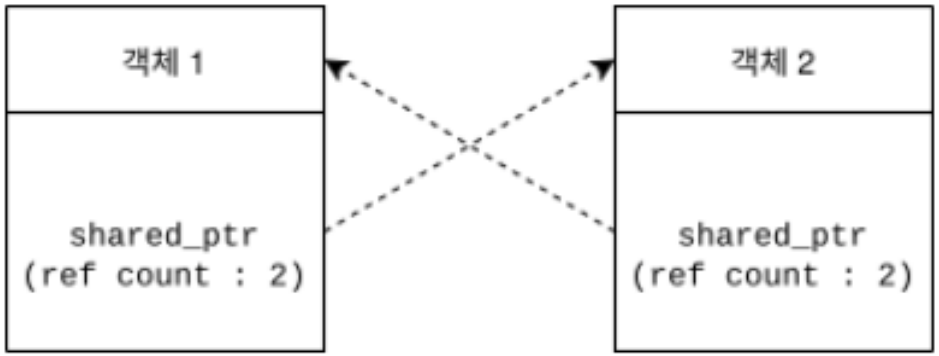
```
Simple* s = new Simple();
shared_ptr<Simple> smartPtr1(s);
shared_ptr<Simple> smartPtr2(s);
```

```
auto smartPtr1 =
    make_shared<Simple>();

shared_ptr<Simple>
    smartPtr2(smartPtr1);
```

weak_ptr

- 리소스를 직접 소유하지 않음.
- shared_ptr 순환 참조 발생 위험 감소



- 삭제될 때, 해당 리소스 삭제 하지 않고, shared_ptr이 삭제 했는지 확인하는 용도로 사용

enable_shared_from_this

- 객체의 method에 shared_ptr 또는 weak_ptr 안전하게 반환 가능함.
  - 객체 포인터가 shared_ptr에 저장되어 있어야 함.

```
class Foo : public
enable_shared_from_this<Foo>
{
public:
    shared_ptr<Foo> getPointer() {
        return shared_from_this();
    }
}

int main()
{
    auto ptr1 = make_shared<Foo>();
    auto ptr2 = ptr1->getPointer();
}
```

```
shared_ptr<Foo> getPointer() {
    return shared_from_this();
}
```

흔히 발생하는 메모리 문제

- 1. 스트링 과소 할당 문제 (*Underallocation*)
  - C++ string 사용
  - 버퍼를 heap 공간에 할당
  - 최대문자 수 (\0 포함) 입력 받아서 초과하면 반환 X
    현재 버퍼 남은 공간과 현재 위치 추적

```
char buf[1024] ={0};
while(true) {
    char* next = getMoreData();
    if (next == nullptr) break;
    else {
        strcat(buf, next);
        delete[] next;
    }
}
```

흔히 발생하는 메모리 문제

- 2. 메모리 경계 침범 (*Buffer Overflow*)
  - 메모리 검사 도구
  - C++ string 또는 vector 사용

```
void FillWithM (char* inStr) {

    int i = 0;
    while (inStr[i] != '\0') {
        inStr[i] = 'm';
        i++;
    }

}
```

흔히 발생하는 메모리 문제

- 3. 메모리 누수
  - mPtr, sPtr unique_ptr 로 생성, outPtr을 unique_ptr 레퍼런스로 !

```
int main()
{
    // 객체 1 생성
    Simple* sPtr = new Simple();
    doSomething(sPtr);
    // 객체 2만 해제함.
    delete sPtr;
    return 0;
}
```

```
class Simple
{
public:
    Simple() {mPtr = new int()}
    ~Simple() {delete mPtr;}
    privaet:
    int* mPtr;
}

void doSomething(Simple*& outPtr)
{
    // bug! 원본 객체 삭제 안함.
    outPtr = new Simple();
}
```

흔히 발생하는 메모리 문제

- 중복삭제와 잘못된 포인터 (Dangling Pointer)
  - 메모리 검사 도구
  - 스마트 포인터 사용
  - 메모리 해제 후 반드시 nullptr 초기화

메모리 누수 탐지 기법

- Visual C++ 메모리 도구
- Valgrind

References

[1] Marc Gregoire, 2018, Professional C++, 4th edition, WILEY