6장 데이터 타입(3)

2020. 6. 8

순천향대학교 컴퓨터공학부

목차

포인터 타입과 참조 타입

- 포인터(pointer) 타입은 메모리 주소와 특정 값, nil로 구성된 값 범위를 갖는 타입

- 포인터 타입의 용도
  - 간접 주소지정
  - 동적 메모리 관리
    - 포인터를 사용하여 동적으로 할당되는 기억공간 접근
    - 동적 할당되는 기억공간을 힙(heap)이라 함
    - 힙으로부터 할당되는 변수는 힙-동적 변수(heap-dynamic variable)
    - 이러한 변수는 이름이 없는 무명 변수(anonymous variable)

포인터 타입 설계 고려사항

- 포인터 변수의 영역과 존속기간은 무엇인가?
- 힙-동적변수의 존속기간은 무엇인가?
- 포인터가 가리킬 수 있는 값의 타입이 제한되는가?
- 포인터의 용도는? 동적 메모리 관리? 간접주소지정? 아니면 이 둘 다인가?
- 언어가 포인터 타입, 참조 타입, 또는 이 두 가지를 모두 지원하는가?

포인터 연산

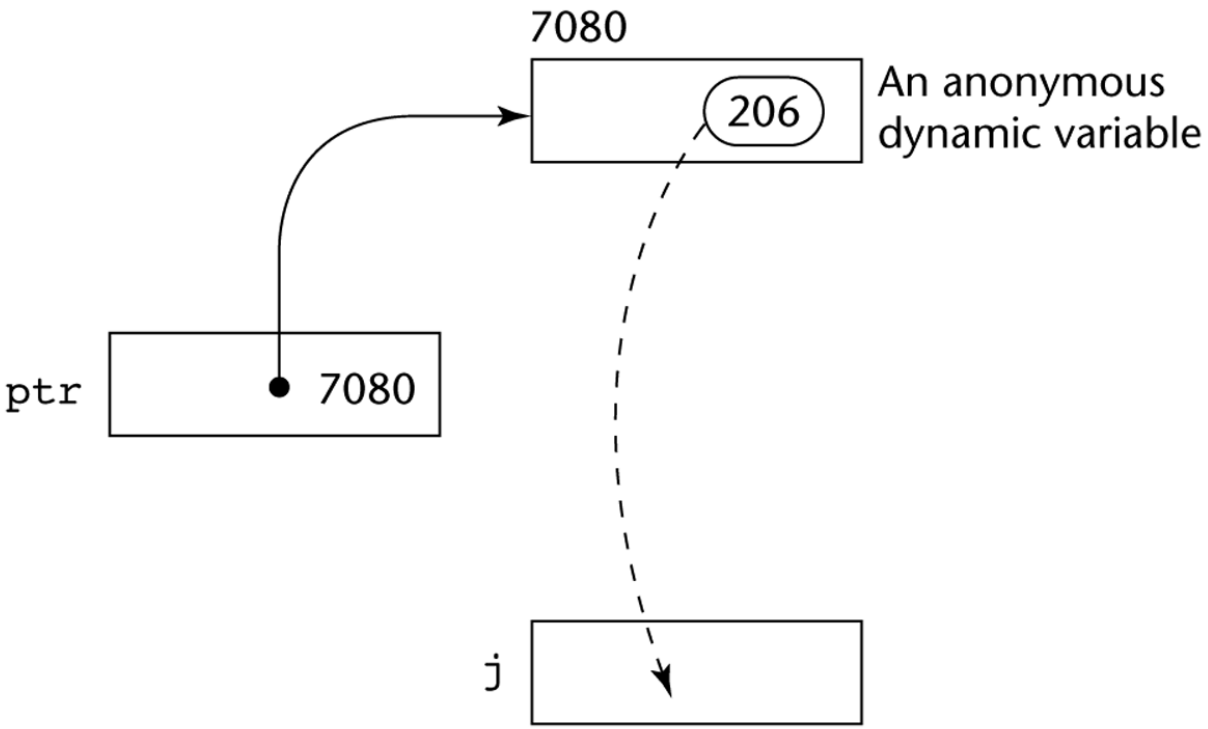
- 기본적인 포인터 연산: 배정, 역참조

- 배정(할당)
  - 포인터 변수에 어떤 유용한 주소로 설정
    - 이 경우 변수 주소를 인출하는 연산자 지원

- 역참조(dereferencing)
  - 포인터가 가리키는 메모리 위치에 저장되어 있는 값을 참조
    - 역참조는 명시적 또는 묵시적(Fortran95+)
    - C, C++는 '*'를 통한 명시적 연산 제공

예제: 포인터 연산

j = *ptr



힙 메모리 관리

- 힙 메모리 관리를 위해 포인터를 제공하는 언어는 명시적 기억공간 할당 연산 제공
  - C: 내장 함수 제공(malloc /free)
  - C++: 할당 연산자 제공(new/ delete)

```
int *data;
…
data = new int[40];
…
delete []data;
```

포인터의 문제

- 허상 포인터
- 분실된 힙-동적 변수

허상 포인터

- 허상 포인터(dangling pointer) 또는 허상 참조(dangling reference)는 이미 회수된 힙-동적 변수를 여전히 가리키는 포인터
  - 허상 포인터의 원인은 힙 동적 변수의 명시적 회수
  - 허상 포인터 생성 과정
    - P1이 힙-동적 변수를 가리키고,
    - P2에 P1을 할당하고,
    - P1이 가리키는 힙-동적 변수를 회수하면?

```
int* arrayPtr1;
int* arrayPtr2 = new int[100];

arrayPtr1 = arrayPtr2;
delete [] arayPtr2;
```

분실된 힙-동적 변수

- 분실된 힙-동적 변수(lost heap-dynamic variables)는 사용자 프로그램에서 더 이상 접근 불가능한 할당된 힙-동적 변수
  - 이러한 힙-동적 변수를 쓰레기(garbage)라 함
  - 분실된 힙-동적변수 생성 과정
    - 포인터 P1이 힙-동적 변수를 가리키도록 설정
    - P1이 나중에 다른 힙-동적 변수를 가리키도록 설정

    => 처음 할당된 힙-동적 변수는 접근 불가, 즉 분실되었음

  - 힙-동적변수 분실을 메모리 누수(memory leakage)라 함

C/C++ 포인터

- 포인터 문제에 대한 해결책을 제공하지 않음
- 제한 없는 포인터 허용으로 극도의 유연성 제공 => 주의 깊게 사용 필요
- 포인터는 임의의 변수를 가리킬 수 있다(변수가 어디에 할당되었는지에 상관없이)
- 포인터는 메모리의 어디든지 가리킬 수 있다(그곳에 변수가 존재하는지의 여부와 상관 없이)

C/C++ 포인터 (2)

- 명시적 역참조(*)와 주소 연산자(&) 제공

```
int *ptr; // 도메인 타입 명세,
          // 올바른 도메인 타입을 갖는 임의 변수 포인트 가능
int count,
…
ptr = &init;
count = *ptr;  // count <- init
```

C/C++ 포인터 (3)

- void* 타입의 포인터는 임의 타입의 값을 가리킬 수 있다 => 포괄형 포인터(generic pointer)라 함
- 이 포인터의 역참조는 불허

```
int a = 10;
void *ptr = &a;

printf("%d", *ptr); // 오류
```

- 사용 예: 메모리 관련 함수의 매개변수
  - 일련의 데이터를 메모리의 한 위치로부터 다른 곳으로 이동하는 함수를 작성할 때

C/C++ 포인터 (4)

- 포인터 산술 연산 허용
  - ptr이 배열을 가리킬 때, 다음 식의 의미는? ptr + index

```
int list[10];
int *ptr;

ptr = list;      // list[0]의 주소가 ptr에 할당

… *(ptr+1)       // list[1]과 동등
… ptr[index]     // list[index]와 동등
            // 배열에 대한 포인터는 배열처럼 인덱싱 가능
```

참조 타입

- 참조 타입(reference type)은 포인터와 유사하나 포인터는 주소를 참조하는 것에 반해, 참조 변수는 메모리의 객체나 값을 참조하는 근본적인 차이
  - 주소에 대한 산술 연산은 자연스러우나, 참조에 대한 산술 연산은 무의미
  - 참조 타입은 제한된 연산을 갖는 포인터

언어의 예

- C++의 참조 타입은 항상 묵시적으로 역참조되는 상수 포인터
  - 변수 정의시 주소 값으로 초기화되어야 하고, 이후에는 다른 변수 참조하는 것으로 변경 불가
  - 주로 함수의 형식 매개변수에서 사용
  - 포인터 매개변수에 비해서 판독성, 안전성 향상

참조 타입 변수

```
int result = 0;
int &ref_result = result;
…
ref_result = 100;
```

실매개변수의 주소가 전달됨

```
swap(int &i, int &j)
{
   int t = i;
   j = j; j=t;
}
```

언어의 예 (2)

- Java
  - C/C++의 포인터 제거
  - 참조 변수는 객체 참조에 제한: 다른 객체 참조 가능
  - 산술 연산 불가
  - 객체는 묵시적으로 회수 (허상 참조 발생가능한가?)

- C#
  - Java의 참조 타입과 C/C++의 포인터를 모두 제공
  - 포인터를 사용하는 메소드는 unsafe 조정자를 포함해야 함 => 포인터 사용은 강력하게 권장되지 않음
  - 참조변수가 가리키는 객체만 묵시적으로 회수되고, 포인터가 가리키는 객체는 그렇지 않음

평가

- 허상 포인터와 분실된 힙-동적 변수 문제
- 힙 메모리 관리 문제
- 포인터는 goto와 같은 것(다음 실행되는 문장범위 vs 참조 가능 메모리 셀 범위)
- 포인터나 참조 변수는 동적 자료구조를 위해서 필수적
- 포인터는 어떤 응용에 필수적 (장치 드라이버 작성 등)
- 참조 변수는 위험성 없이 포인터의 어느 정도의 유연성과 능력 제공

포인터와 참조 타입 구현

- 포인터와 참조 표현: 셀에 포함된 한 개의 주소나 (세그먼트, 오프셋)
- 허상 포인터 문제 해결책
- 힙 메모리 관리

허상 포인터 문제 해결책

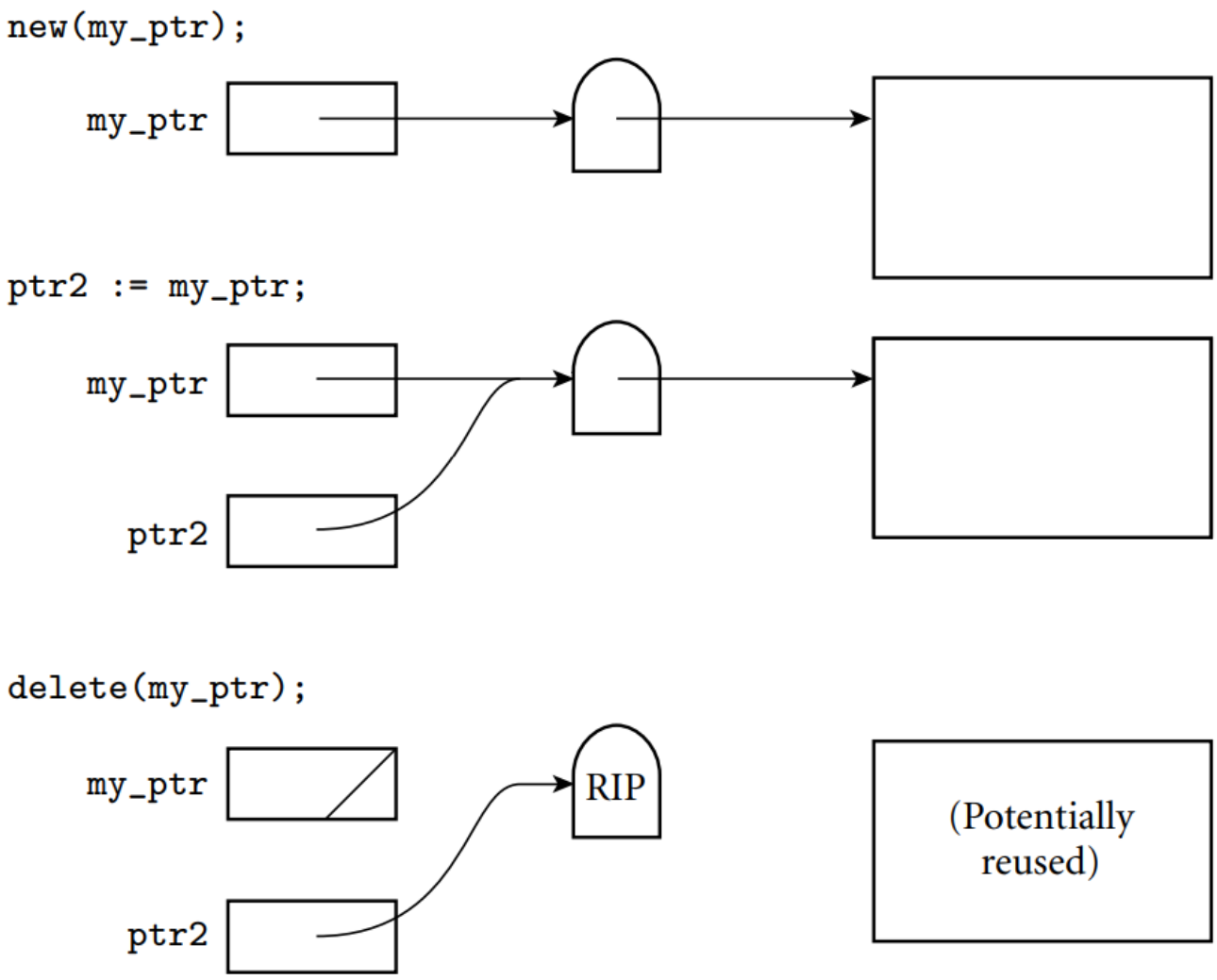
- 비석 접근방법(tombstone)
- 잠금-키 접근방법 (lock-and-key approach)

비석 접근방법

- 비석 접근 방법(tombstone)
  - 힙-동적 변수는 여분의 힙 셀인 비석을 포함
  - 비석은 힙-동적 변수에 대한 포인터
  - 실제의 포인터 변수는 단지 비석만을 가리키고,
  - 힙-동적 변수가 회수될 때, 비석은 남아 있고 nil로 설정(이는 그 변수가 회수되었음을 나타냄)

- 시간과 공간의 관점에서 비용 부담
  - 비석은 회수되지 않음
  - 비석을 통한 힙-동적변수에 대한 접근은 한 레벨 이상 간접 참조

비석 접근 방법 (2)

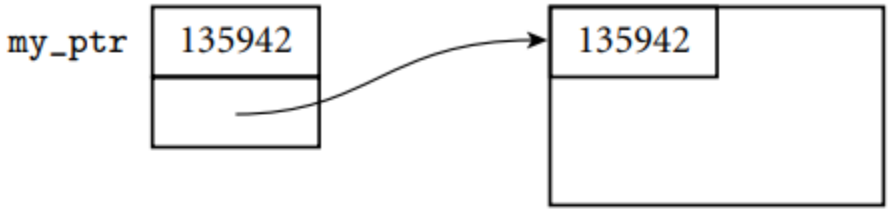

new(my_ptr);
my_ptr
ptr2 := my_ptr;
my_ptr
ptr2
delete(my_ptr);
my_ptr
ptr2
RIP
(Potentially reused)

잠금-키 접근방법

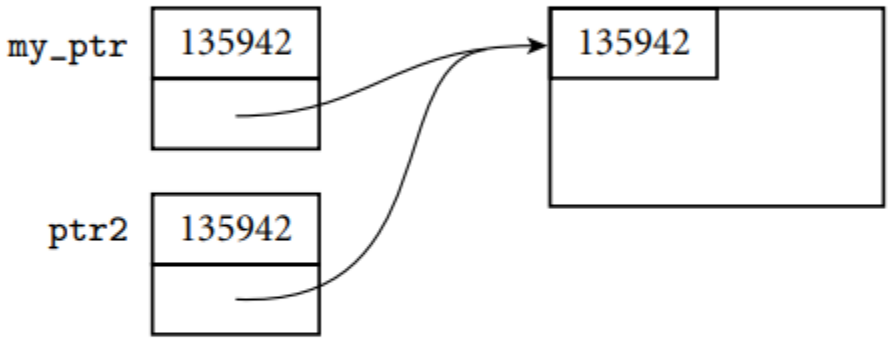
- 잠금-키 접근방법 (lock-and-key approach)
  - 포인터 값은 (키, 주소)의 순서쌍으로 표현
  - 힙-동적 변수는 변수와 잠금 셀(정수의 잠금 값 포함)로 표현
  - 힙-동적 변수가 할당될 때, 잠금 값이 생성되고, 그 변수의 잠금 셀과 포인터의 키 셀에 모두 저장됨
  - 역참조된 포인터의 모든 접근은 이 포인터의 키 값과 힙-동적 변수의 잠금 값과 비교. 이 두 개 값이 일치한 경우에만 그 접근은 적법하다.
  - 힙-동적 변수가 회수될 때, 잠금 값은 적법하지 않은 값으로 변경

잠금-키 접근 방법 (2)

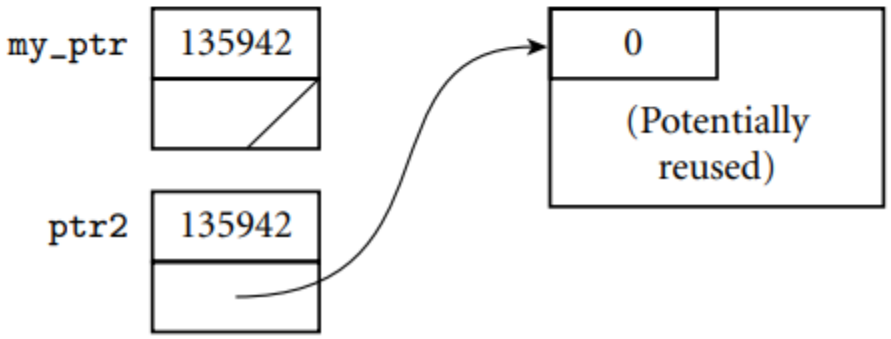
```
new(my_ptr);
```




```
ptr2 := my_ptr;
```




```
delete(my_ptr);
```



힙 메모리 관리

- 매우 복잡한 실행-시간 프로세스
- 할당/회수 단위: 단일-크기 셀 vs. 가변 크기 셀
- 단일-크기 할당 힙 메모리 관리
  - 모든 유용 가능한 셀들은 셀에 포함된 포인터들로 서로 연결되어 가용 공간 리스트 구성
  - 할당은 이 리스트로부터 요구된 셀의 개수를 가져오는 것
- 힙-동적 변수 회수는 훨씬 더 복잡한 프로세스
  - 힙 동적-변수 회수 여부를 결정하는 것이 어렵다
  - 셀의 회수를 누가하는가? 프로그래머 혹은 시스템
  - 시스템에 의해서 회수된다면 언제할 것인가?

힙 메모리 관리

- 쓰레기를 회수하는 2가지 접근방법
  - 참조 계수기(reference counter)
    - 조기 접근 방법(eager approach)
    - 접근될 수 없는 셀 발생시 회수(회수는 점차적으로)
  - 표시-수집(mark-sweep)
    - 지연 접근 방법(lazy approach)
    - 가용 기억 공간이 바닥날 때 회수

참조 계수기

- 모든 셀에 참조 계수기를 유지하고, 이 계수기에는 현재 셀을 가리키는 포인터의 개수를 저장
  - 포인터가 셀을 새롭게 가리키면 참조 계수기를 증가
  - 포인터가 셀에서 분리될 때 참조 계수기 값을 감소하고, 이때 그 값이 0이면 그 셀은 쓰레기이고, 가용 공간리스트에 반환

- 단점:
  - 기억 공간 부담
  - 계수기 값 유지에 따른 실행시간 부담
  - 셀들의 모임이 순환적으로 연결되어 있을 경우 복잡성

- 장점:
  - 점차적 회수로 프로그램 실행에 심각한 지연 회피

표시-수집

- 실행-시간 시스템은 셀을 할당하고, 포인터로부터 셀들을 분리셀들을 분리(회수를 고려하지 않음)

- 가용 기억공간이 바닥날 때 표시-수집 과정이 시작
  - 모든 힙 셀은 수집 알고리즘에서 사용되는 여분의 비트를 갖는다
  - 모든 셀의 비트는 초기에 쓰레기로 설정
  - 힙 공간을 가리키는 모든 포인터들을 추적하여 그 포인터로부터 접근 가능한 셀들은 쓰레기가 아니라고 표시
  - 모든 쓰레기 셀들을 가용 공간 리스트에 반환

- 단점:
  - 표시-수집 과정이 너무 드물게 수행되고, 수행시 프로그램 실행에 심각한 지연 초래
  - 점진적 표시-수집(incremental mark-sweep)으로 개선

가변-크기 셀

- 단일-크기 셀의 힙 관리에 또다른 문제 고려

- 표시-수집 방법 사용시
  - 모든 셀을 쓰레기로 초기 설정하는 것이 어렵다
  - 표시 과정도 간단하지 않다
  - 가용 기억공간 리스트 유지에 부담
  - 리스트가 결국 매우 작은 크기의 블럭들을 포함하게 되어 인접 블럭들을 합병하는 것이 필요