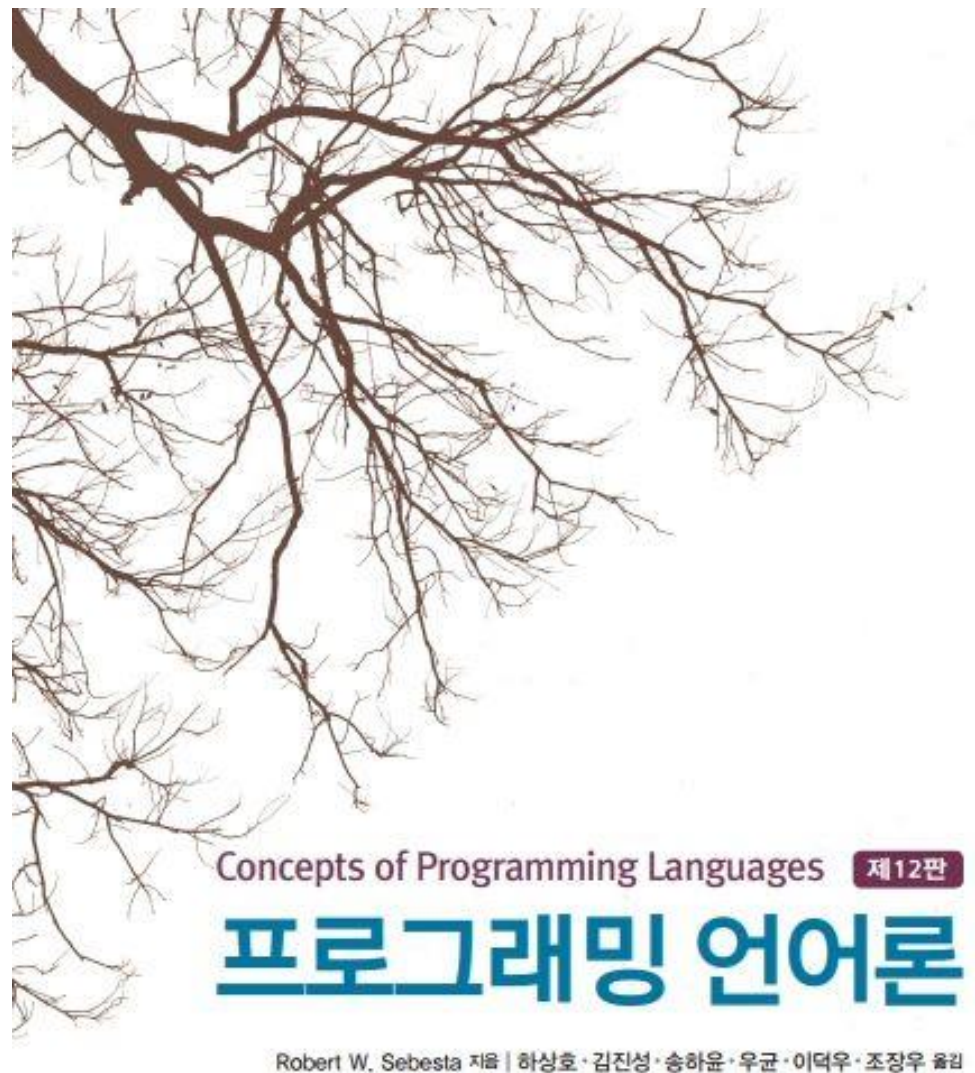


## 6장

## 데이터 타입



# 주제

---

- 서론
- 기본 데이터 타입
- 문자 스트링 타입
- 열거 타입
- 배열 타입
- 연관 배열
- 레코드 타입
- 튜플 타입
- 리스트 타입
- 공용체 타입
- 포인터 타입과 참조 타입
- 선택적 타입
- 타입 검사
- 강 타입
- 타입 동등
- 이론과 데이터 타입

# 포인터 타입과 참조 타입

---

- 포인터(pointer) 타입은 값 범위로 메모리 주소와 특수값 **nil**을 갖는 타입
- 포인터 타입의 용도
  - 간접 주소 지정
  - 동적 메모리 관리
- 용어
  - 동적 할당되는 기억공간을 **힙(heap)**이라 함
  - 힙으로부터 할당되는 변수는 **힙-동적 변수(heap-dynamic variable)**
  - 힙-동적 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)
  - 동적 할당된 기억공간의 주소를 가리키거나
  - 다른 변수를 간접 참조(이 경우 변수 주소를 인출하는 명시적 연산자가 지원되어야 함)
- 역참조(dereferencing)
  - 포인터가 가리키는 메모리 위치에 저장되어 있는 값을 참조
  - 역참조는 명시적 또는 묵시적
  - C, C++는 '\*'를 통한 명시적 연산 제공

# 예제: 포인터 연산

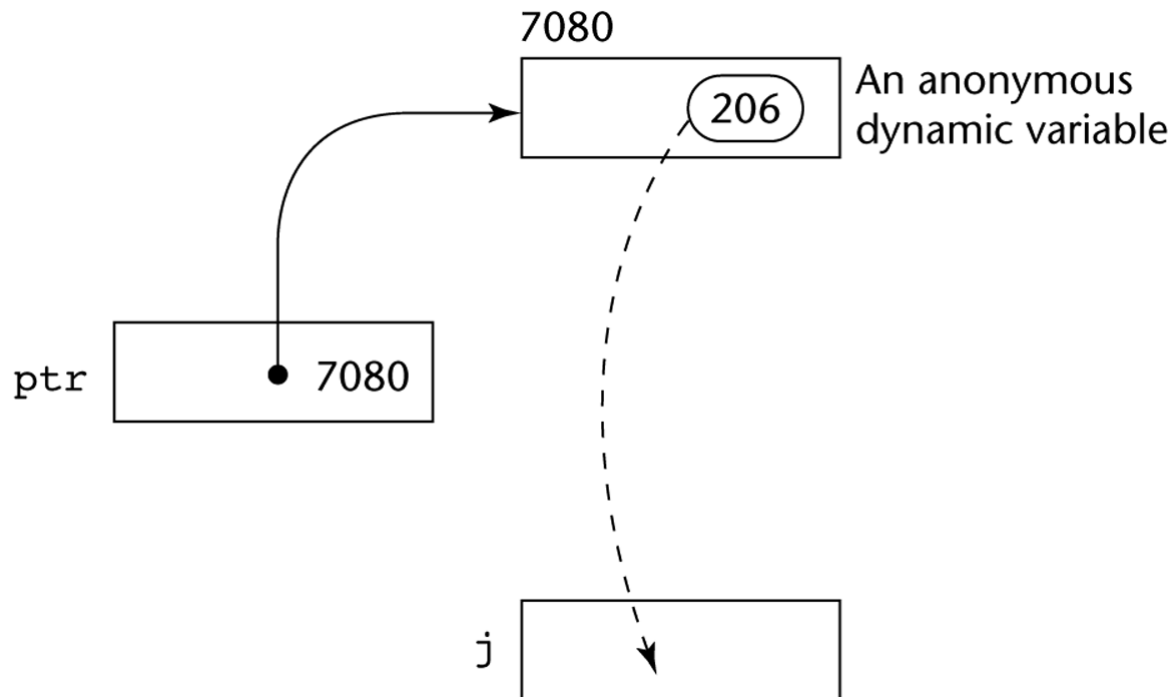
```
int j, *ptr;
```

```
// void* malloc(size_t size);
```

```
ptr = (int *)malloc(sizeof(int));
```

```
*ptr = 206;
```

```
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

---

- 포인터는 PL/1에서 처음으로 도입
  - 높은 유연성, 그러나 상당한 프로그래밍 오류 초래
- Java는 포인터 연산을 제한한 참조 타입으로 대체
  - 참조타입은 제한된 연산을 갖는 포인터
- 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

```
#include <iostream>
```

```
int* createPointer() {  
    int local = 42;  
    return &local; // 지역변수 주소  
}
```

```
int main() {  
    int* p = createPointer();  
    *p = 20;  
    std::cout << *p << std::endl; // 허상포인터  
    return 0;  
}
```

```
#include <iostream>
```

```
int main() {  
    int* ptr = new int(100);  
    delete ptr;    // 메모리 해제  
    *ptr = 100;    // 허상 포인터  
    std::cout << *ptr; // 허상 포인터  
    return 0;  
}
```

# 분실된 힙-동적 변수

---

- 분실된 힙-동적 변수(lost heap-dynamic variables)는 사용자 프로그램에서 더 이상 접근 불가능한 할당된 힙-동적 변수
- 분실된 힙-동적변수 생성 과정
  - 포인터 P1 이 힙-동적 변수를 가리키도록 설정
  - P1 이 나중에 다른 힙-동적 변수를 가리키도록 설정
- 이러한 힙-동적 변수를 **쓰레기**(garbage)라 하고, 힙-동적변수 분실을 **메모리 누수**(memory leakage)라 함

# C/C++ 포인터

---

- 명시적 역참조(\*)와 주소 연산자(&) 제공
- 포인터 선언은 그 데이터 타입을 명시
- 포인터는 선언 시 명시된 데이터 타입을 갖는 변수를 가리킬 수 있다.

```
int *ptr; // 데이터 타입 명시
int count,
...
ptr = &init;
Count = *ptr; // count = init
```

# C/C++ 포인터 산술 연산

---

- 포인터 산술 연산은 주로 배열 조작에 사용
- 배열에 대한 포인터는 인덱싱 가능

```
int list[10]; // 배열 이름은 첫번째 요소를 참조
int *ptr;

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

... *(ptr+1)    // list[1]과 동등
... ptr[index]  // list[index]와 동등
```

# C/C++의 void\* 포인터

- void\* 타입의 포인터는 임의 타입의 값을 가리킬 수 있다(포괄형 포인터(generic pointer))

- 이 포인터의 역참조는 불허

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

printf("%d", *ptr); // 오류
```

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

# C/C++ 포인터 평가

---

- 제한 없는 포인터 허용으로 극도의 유연성을 제공하나 주의 깊게 사용 필요
  - 포인터는 메모리의 어디든지 가리킬 수 있다.
- 허상포인터와 분실된 힙-동적변수의 포인터 문제에 대한 해결책을 제공하지 않음

# 참조 타입

---

- **참조 타입**(reference type)은 포인터와 유사하나 근본적인 차이점:
  - 포인터는 메모리의 주소를 참조하는 것에 반해, 참조 변수는 메모리의 객체나 값을 참조
  - 그 결과, 참조에 대한 산술 연산은 무의미 (주소에 대한 산술 연산은 자연스러움)


# C++ 참조 타입

---

- 항상 묵시적으로 역참조되는 상수 포인터
  - 참조 변수 선언시 주소 값으로 초기화되어야 하고, 이후에는 다른 변수 참조 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;  
}
```





# Java의 참조 타입

---

- C/C++의 포인터 제거
- 참조 변수의 유일한 용도는 객체에 대한 참조
- 다른 클래스 객체도 참조 가능(상수가 아님)
- 허상 참조가 발생할 수 있는가?

# C#의 참조 타입

- Java의 참조 타입과 C/C++의 포인터를 모두 제공
  - 포인터를 사용하는 메소드는 **unsafe** 조정자를 포함해야 함 (포인터 사용이 권장되지 않음)
  - 참조 변수가 가리키는 객체만 묵시적으로 회수
- 
- Python은?

# 평가

---

- 포인터는 goto와 같은 것
  - 다음 실행 가능한 문장의 범위, 참조 가능한 메모리 셀의 범위를 확장
- 포인터는 어떤 응용에 필수적
  - 디바이스 드라이버 작성 등 (ex. memory mapped IO 장치)
- 참조 변수는 위험성 없이 포인터의 어느 정도 유연성과 기능 제공

# 타입 검사

---

- **타입 검사**(type checking)는 연산자에 대한 피연산자들의 타입이 호환가능한지를 확인하는 행위
- 피연산자와 연산자의 개념에 부프로그램과 배정문을 포함시킴
- **호환가능 타입**(compatible type)이란 연산자에 대해서 적법하거나, 또는 적법한 타입으로 컴파일러에 의해서 묵시적으로 변환되는 타입
- 타입의 묵시적 자동변환을 **타입 강제변환**(coercion)이라 한다.
- **타입 오류**(type error)는 연산자를 부적절한 타입의 피연산자에 적용할 때 발생

# 타입 검사

---

- 모든 타입 바인딩이 정적이면, 거의 모든 타입 검사는 정적일 수 있다. 이를 **정적 타입 검사**(static type checking)라 한다.
- 타입 바인딩이 동적이면, 타입 검사는 동적이어야 하는데, 이를 **동적 타입 검사**(dynamic type checking)이라 한다.
- 효율성, 신뢰성, 비용 vs 유연성

# 강 타입

---

- 프로그래밍 언어는 타입 오류가 항상 탐지되면 **강 타입** (strongly typed languages)이다.
  - 모든 피연산자의 타입이 컴파일 시간 또는 실행 시간에 결정
  - 타입 오류를 초래하는 변수의 모든 잘못된 사용 탐지
- C, C++
  - 강타입 언어가 아님. Why?
- Java, C#
  - 강타입 언어
  - 명시적 타입 캐스팅에 의한 오류 가능성.

# 타입 강제 변환

---

- 타입 강제 변환은 강 타입을 약화시킴

```
int a, b;  
float d;  
...  
a = a+d; // a + b의 작성을 의도했다면?
```

- 다음 언어에서 타입 강제 변환의 정도는?
  - C, C++
  - Java, C#
  - ML, F#

# 타입 동등(type equivalence)

---

- 배열과 레코드의 구조화된 타입에 대해서는  
**타입 동등**(type equivalence) 개념 적용
  - 식에서 한 타입의 피연산자가 타입 강제 변환 없이 다른 타입의 피연산자로 대체될 수 있으면, 이 두 개의 타입은 동등하다.
- 타입 동등을 정의하는 2가지 접근 방법
  - 이름 타입 동등(name type equivalence)
  - 구조 타입 동등(structure type equivalence)



# 이름 타입 동등

---

- 두 변수가 동일한 선언문에 속하거나, 같은 타입 이름으로 선언되면, 이러한 두 변수는 **이름 타입 동등**(name type equivalence)하다.
- 구현하기 쉬우나 매우 제약적

```
type idxtype 1..100; // 새로운 타입 생성
count: integer;
idx: idxtype; // idx와 count는 타입 동등한가?
...
count = idx + count // 타입 오류?
```

# 구조 타입 동등

---

- 두 변수의 타입이 동일한 구조를 가지면 **구조 타입 동등**(structure type equivalence)하다.
- 더 유연하지만 구현하기 어렵다
  - 두 타입의 전체 구조를 비교하는 것이 간단하지 않다.
    - 두 레코드 타입이 구조적으로 동일하나 다른 필드 이름을 가지면
    - 두 배열이 첨자만이 다른 경우 (e.g. [1..10], [0..9])
  - 동일한 구조를 갖는 타입들을 구별할 수 없다.

```
type celsius = real;  
type fahrenheit = real; // celsius와 fahrenheit는 타입 동등?
```

# 언어 예제: C/C++

---

- `struct`, `union`, `enum` 선언은 새로운 타입을 생성하여 이름 타입 동등이 적용되고, 다른 구조화된 타입(배열, 포인터)에는 구조 타입 동등이 적용
- `typedef`는 새로운 타입을 생성하지 않으며, 단지 기존 타입에 대한 새로운 이름을 부여 (`typedef`로 정의된 타입은 그 부모 타입과 타입 동등함)
- `struct`, `union`, `enum`이 다른 파일에 정의되면 구조 타입 동등이 적용됨(loophole, C++에서는 이러한 예외사항이 없음)

# 언어 예제: C/C++ (계속)

---

- 다음에서 struct A, struct B, C는 동등한가?

```
struct A { // 새로운 타입
    char x;
    int y;
};
struct B {
    char x;
    int y;
};
typedef struct A C;
```