Structures, Unions, and Enumerations

adopted from KNK C Programming : A Modern Approach

Structure (구조체) Variables

- The properties of a ***structure*** are different from those of an array.
  구조체는 배열과 다름
  - The elements of a structure (its ***members***) aren't required to have the same type.
    구조체의 구성요소(멤버)는 서로 다른 형을 갖을 수 있음
  - The members of a structure have names; to select a particular member, we specify its name, not its position.
    구조체의 멤버는 이름을 갖고 있고, 그 멤버를 사용하기위해선 이름을 사용함
    (비교: 배열은 인덱스 번호로 활용)
- In some languages, structures are called ***records,*** and members are known as ***fields.***
  어떤 언어는 구조체를 레코드, 멤버는 필드라고 부름

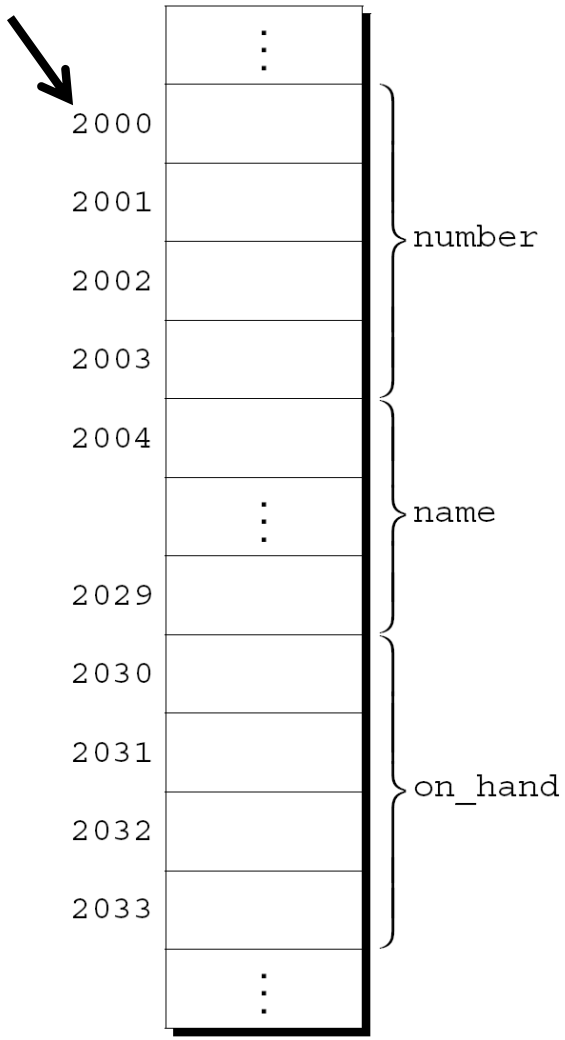
Declaring Structure Variables

- A structure is a logical choice for storing a collection of related data items.
  구조체는 관련있는 데이터를 묶을 수 있는 틀을 제공
- A declaration of two structure variables that store information about parts in a warehouse:
  창고에 있는 부품 정보를 저장하기 위해 두 개의 구조체 변수를 선언하는 예

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

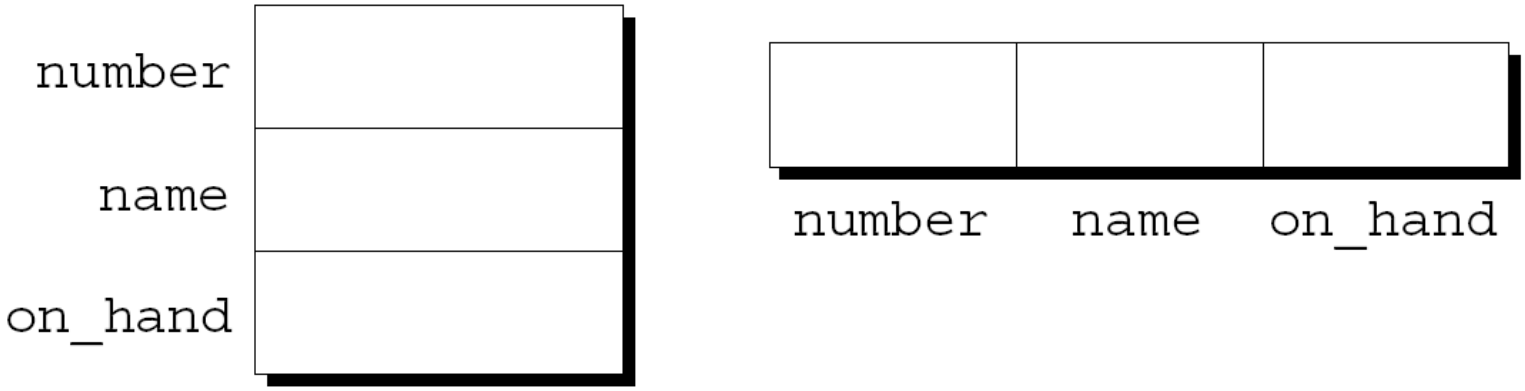
Declaring Structure Variables

- The members of a structure are stored in memory in the order in which they're declared.
  구조체의 멤버는 선언된 순서대로 메모리에 배치됨
- Appearance of `part1` (메모리에 표현된 모습)
- Assumptions(가정):
  - `part1` is located at address 2000.
    시작 주소 2000
  - Integers occupy four bytes.
    정수는 4바이트
  - `NAME_LEN` has the value 25. NAME_LEN = 25
  - There are no gaps between the members.
    멤버들 사이에는 공백이 없음



Declaring Structure Variables

- Abstract representations of a structure:
  추상화된 구조체의 표현



- Member values will go in the boxes later.
  실제 값은 이후에 저장됨

Declaring Structure Variables

- Each structure represents a new scope.
  각 구조체는 새로운 범위를 갖고 있음
- Any names declared in that scope won't conflict with other names in a program.
  구조체 내에서 선언된 이름은 프로그램 다른 곳에서 사용된 이름과 충돌하지 않음
- In C terminology, each structure has a separate ***name space*** for its members.
  각 구조체의 멤버는 독립적인 이름공간을 갖음

Declaring Structure Variables

- For example, the following declarations can appear in the same program:
  예를 들어 아래와 같이 두 개의 구조체가 한 프로그램에서 선언됨
  number, name 의 이름이 같지만 충돌하지 않음

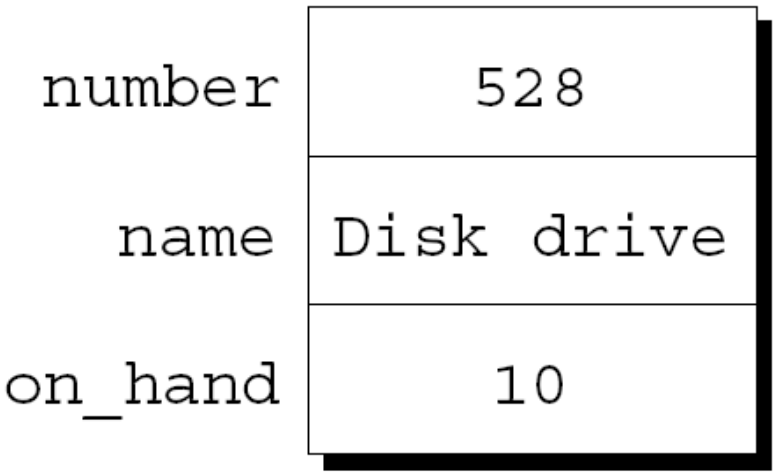
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```

Initializing Structure Variables

- A structure declaration may include an initializer:
  구조체 선언 시 초기화할 수 있음

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization: 초기화 후 모습



Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers.
  구조체 초기화하는 배열의 초기화와 유사한 규칙을 따름
- Expressions used in a structure initializer must be constant. (This restriction is relaxed in C99.)
  구조체 초기화에 사용한 수식은 상수여야 함(c99에서는 규칙이 완화됨)
- An initializer can have fewer members than the structure it's initializing.
  구조체의 전체 멤버보다는 적은 수의 멤버를 초기화 할 수 있음
- Any "leftover" members are given 0 as their initial value.
  초기화에 제외된 멤버는 0으로 초기화 됨

Designated Initializers (C99)

- C99's designated initializers can be used with structures.
  c99는 구조체의 멤버를 지정하여 지정 초기화 가능
- The initializer for `part1` shown in the previous example:
  part1의 일반적 초기화는 다음과 같음

```
{528, "Disk drive", 10}
```

- In a designated initializer, each value would be labeled by the name of the member that it initializes:
  지정 초기화하는 다음과 같이 멤버의 이름을 지목하여 초기화 할 수 있음

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

- The combination of the period and the member name is called a ***designator.***
  지정하는 방법: .(점) + 멤버이름 예, .number

Designated Initializers (C99)

- Designated initializers are easier to read and check for correctness.
  지정 초기화를 하면 값을 쉽고 정확함
- Also, values in a designated initializer don't have to be placed in the same order that the members are listed in the structure.
  지정 초기화를 하면 구조체에 나온 멤버의 순서대로 나열하지 않아도 됨
  - The programmer doesn't have to remember the order in which the members were originally declared.
    프로그래머는 멤버의 순서를 기억할 필요 없음
  - The order of the members can be changed in the future without affecting designated initializers.
    멤버의 순서가 시간이 지나 바뀌더라도 지정 초기화의 순서는 바뀌지 않아도 됨

Designated Initializers (C99)

- Not all values listed in a designated initializer need be prefixed by a designator.
  지정 초기화할 때 멤버이름을 써야 하는 것은 아님
- Example:

  ```
  {.number = 528, "Disk drive", .on_hand = 10}
  ```

  The compiler assumes that "`Disk drive`" initializes the member that follows `number` in the structure.
  ”disk drive”는 number 다음에 나오는 멤버의 값이라고 가정함
- Any members that the initializer fails to account for are set to zero.
  초기화가 실패한 경우 0으로 초기화

Operations on Structures

- To access a member within a structure, we write the name of the structure first, then a period, then the name of the member.
  구조체의 멤버를 접근하기 위해 구조체의 이름, 점, 그리고 멤버 이름을 씀
- Statements that display the values of `part1`'s members:
  part1이라는 구조체의 멤버를 읽어 오는 방법 예

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Operations on Structures

- The members of a structure are lvalues.
  구조체의 멤버는 lvalue임
- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:
  구조체의 멤버는 할당문에 왼쪽에 올 수 있고 증감 연산자도 사용할 수 있음

```
part1.number = 258;
  /* changes part1's part number */
part1.on_hand++;
  /* increments part1's quantity on hand */
```

Operations on Structures

- The period used to access a structure member is actually a C operator.
  점은 c에서 연산자이고 구조체의 멤버를 접근할 수 있도록 함
- It takes precedence over nearly all other operators.
  점은 거의 대부분의 연산자보다 우선순위가 높음
- Example:

```
scanf("%d", &part1.on_hand);
```

  The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`.
  & 연산자보다 점의 우선순위가 높음 &part1.on_hand라고 하면 part1.on_hand의 주소임

Operations on Structures

- The other major structure operation is assignment:
  구조체의 할당 예

```
part2 = part1;
```

- The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.
  part1 구조체의 모든 멤버의 값을 part2의 모든 멤버로 복사

Operations on Structures

- Arrays can't be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied.
  배열은 = 연산자로 복사 안됨
- Some programmers exploit this property by creating "dummy" structures to enclose arrays that will be copied later:
  어떤 프로그래머는 구조체의 기능을 이용해 배열의 값 복사에 활용하기도 함

```
struct { int a[10]; } a1, a2;
a1 = a2;
  /* legal, since a1 and a2 are structures */
```

Operations on Structures

- The = operator can be used only with structures of ***compatible*** types. =연산자는 호환가능한 구조체에서만 활용 가능
- Two structures declared at the same time (as `part1` and `part2` were) are compatible. 동시에 선언된 구조체 변수들만 호환가능
- Structures declared using the same "structure tag" or the same type name are also compatible. 같은 구조체 태그를 써서 선언된 변수도 호환됨
- Other than assignment, C provides no operations on entire structures. 할당 외에는 구조체 전부를 다루는 연산자는 없음
- In particular, the == and != operators can't be used with structures. 예를 들어 ==나 != 연산자는 쓸 수 없음

Structure Types

- Suppose that a program needs to declare several structure variables with identical members.
  같은 멤버를 갖는 구조체 변수를 여러 개 써야 한다고 가정해보자
- We need a name that represents a *type* of structure, not a particular structure *variable*.
  구조체를 형으로 쓰기 위해서는 이름이 필요함, 구조체 변수 이름이 아님
- Ways to name a structure: 구조체에 이름을 부여하기
  - Declare a "structure tag" 구조체 태그를 선언
  - Use `typedef` to define a type name typedef으로 형의 이름 저정

Declaring a Structure Tag

- A ***structure tag*** is a name used to identify a particular kind of structure.
  구조체 태그는 구조체들을 구분하기 위한 이름
- The declaration of a structure tag named `part`:
  part라는 이름의 구조체 태그의 선언

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Note that a semicolon must follow the right brace.
  오른쪽 끝에 세미콜론이 붙었음

Declaring a Structure Tag

- The `part` tag can be used to declare variables:
  이제 part 태그를 써서 변수들을 선언할 수 있음

```
struct part part1, part2;
```

- We can't drop the word `struct`:

```
part part1, part2;    /*** WRONG ***/
```

  `part` isn't a type name; without the word `struct`, it is meaningless.
  만약 struct라는 단어를 안쓰면 오류가 생김; part는 구조체 구별 용 태그 이름일 뿐

- Since structure tags aren't recognized unless preceded by the word `struct`, they don't conflict with other names used in a program.
  struct 없는 태그 이름은 의미 없음; 프로그램 내에 다른 것의 이름으로 쓸 수 있음

Declaring a Structure Tag

- The declaration of a structure *tag* can be combined with the declaration of structure *variables:*
  구조체 태그와 구조체 변수의 이름을 같이 쓸 수 있음

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Declaring a Structure Tag

- All structures declared to have type `struct part` are compatible with one another:
  struct part라는 태그를 갖는 구조체로 선언된 변수는 모두 호환 가능

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;
  /* legal; both parts have the same type */
```

Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name.
  구조체 태그를 선언하는 대신 typedef으로 새로운 타입을 지정할 수 있음
- A definition of a type named `Part`: Part 형의 정의

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- `Part` can be used in the same way as the built-in types:
  다른 기본 형들과 같이 Part 형도 쓸 수 있음

```
Part part1, part2;
```

Defining a Structure Type

- When it comes time to name a structure, we can usually choose either to declare a structure tag or to use `typedef`.
  구조체의 이름을 정할 때는 구조체 태그 또는 typedef를 쓸 수 있음
- However, declaring a structure tag is mandatory when the structure is to be used in a linked list (Chapter 17).
  단, 연결리스트(linked list)를 쓸 때는 꼭 구조체 태그를 사용해야 함

Structures as Arguments and Return Values

- Functions may have structures as arguments and return values.
  함수 리턴 값과 인자로 구조체 사용 가능
- A function with a structure argument: 구조체를 인자로 갖는 함수 예

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- A call of `print_part`: 호출 예

```
print_part(part1);
```

Structures as Arguments and Return Values

- A function that returns a `part` structure:
  part 구조체를 리턴하는 함수

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

- A call of `build_part`: 호출 예

```
part1 = build_part(528, "Disk drive", 10);
```

Structures as Arguments and Return Values

- Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure.
  구조체를 함수 인자로 전달 또는 리턴 값으로 받으면, 구조체 모든 멤버가 복사됨
- To avoid this overhead, it's sometimes advisable to pass a pointer to a structure or return a pointer to a structure.
  복사비용을 줄이려면 구조체에 대한 포인터를 인자 또는 리턴 값으로 사용
- Chapter 17 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure.
  17장에 활용 예가 있음

Structures as Arguments and Return Values

- There are other reasons to avoid copying structures.
  구조체 복사를 피해야할 또 다른 이유가 있음
- For example, the `<stdio.h>` header defines a type named `FILE`, which is typically a structure.
  <stdio.h> 헤더 정보에는 FILE이라는 구조체가 정의됨
- Each `FILE` structure stores information about the state of an open file and therefore must be unique in a program.
  각 FILE 구조체는 열어놓은 파일의 상태정보를 갖고 있음; 모든 프로그램에서 FILE의 정보는 유일해야함
- Every function in `<stdio.h>` that opens a file returns a pointer to a `FILE` structure.
  <stdio.h> 내의 함수들 중 파일 열기를 하는 함수들은 모두 FILE 구조체의 포인터를 리턴함
- Every function that performs an operation on an open file requires a `FILE` pointer as an argument.
  열어 놓은 파일을 조작하는 함수들은 FILE의 포인터를 인자로 받음

Structures as Arguments and Return Values

- Within a function, the initializer for a structure variable can be another structure:

```
void f(struct part part1)
{
  struct part part2 = part1;
  …
}
```

- The structure being initialized must have automatic storage duration.

Compound Literals (C99)

- Chapter 9 introduced the C99 feature known as the ***compound literal.***
  복합 문자열을 9장에서 살펴봄
- A compound literal can be used to create a structure "on the fly," without first storing it in a variable.
  복합 문자열을 쓰면 즉석해서 구조체를 만들어 값을 저장할 수 있음
- The resulting structure can be passed as a parameter, returned by a function, or assigned to a variable.
  이렇게 생성된 구조체는 매개변수로 쓰이거나 함수의 리턴 또는 변수에 할당될 수 있음

Compound Literals (C99)

- A compound literal can be used to create a structure that will be passed to a function:

```
print_part((struct part) {528, "Disk drive", 10});
```

The compound literal is shown in **bold**.
복합 문자열을 써서 함수에 전달할 구조체를 만들 수 있음; 굵은 글씨가 예

- A compound literal can also be assigned to a variable:

```
part1 = (struct part) {528, "Disk drive", 10};
```
변수 할당에도 활용할 수 있음

Compound Literals (C99)

- A compound literal consists of a type name within parentheses, followed by a set of values in braces.
  복합 문자열은 괄호로 묶인 형의 이름과 중괄호로 묶인 값들의 집합으로 구성
- When a compound literal represents a structure, the type name can be a structure tag preceded by the word `struct` or a `typedef` name.
  복합 문자열이 구조체를 나타낸다면, 형의 이름은 struct 키워드와 구조체 태그가 있거나 typedef 이름으로 정의된 것이어야 함

Compound Literals (C99)

- A compound literal may contain designators, just like a designated initializer:
  복합 문자열은 지정초기화될 수 있음

```
print_part((struct part) {.on_hand = 10,
                          .name = "Disk drive",
                          .number = 528});
```

- A compound literal may fail to provide full initialization, in which case any uninitialized members default to zero.
  복합 문자열 선언으로 일부만 초기화할 경우, 초기화 안된 멤버는 0으로 초기화됨

Nested Arrays and Structures

- Structures and arrays can be combined without restriction.
  구조체와 배열은 제한 된 조건으로 병합될 수 있음
- Arrays may have structures as their elements, and structures may contain arrays and structures as members.
  배열은 구조체를 요소로 갖을 수 있음, 구조체는 배열뿐만 아니라 구조체도 멤버로 갖을 수 있음

Nested Structures

- Nesting one structure inside another is often useful.
  한 구조체를 다른 구주체의 일부로 쓰는 것은 유용함
- Suppose that `person_name` is the following structure:
  person_name이라는 구조체를 살펴보자

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

Nested Structures

- We can use `person_name` as part of a larger structure:
  person_name을 다른 큰 구조체의 일부로 쓸 수 있음

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Accessing `student1`'s first name, middle initial, or last name requires two applications of the . operator:
  student1의 이름(first, middle, last)를 접근하기 위해서는 점 연산자를 두번 사용

```
strcpy(student1.name.first, "Fred");
```

Nested Structures

- Having `name` be a structure makes it easier to treat names as units of data.
  name은 구조체이기 때문에 이름을 하나의 데이터 단위로 처리 가능
- A function that displays a name could be passed one `person_name` argument instead of three arguments:
  person_name을 인자로 쓸 때는 변수 3개가 아니라 하나면 됨

```
display_name(student1.name);
```

- Copying the information from a `person_name` structure to the `name` member of a student structure would take one assignment instead of three: 매개변수로 복사 될 때도 한번으로 복사 가능

```
struct person_name new_name;
…
student1.name = new_name;
```

Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
  배열과 구조체의 혼합의 가장 흔한 조합은 배열의 요소가 구조체인 경우
- This kind of array can serve as a simple database.
  배열을 데이터베이스처럼 쓸 수 있음
- An array of `part` structures capable of storing information about 100 parts:
  part 구조체를 배열로 선언한 경우, 배열의 크기만큼 자료를 저장할 수 있음

```
struct part inventory[100];
```

Arrays of Structures

- Accessing a part in the array is done by using subscripting:
  part 배열 중 하나를 접근하려면 첨자를 쓰면 됨

```
print_part(inventory[i]);
```

- Accessing a member within a `part` structure requires a combination of subscripting and member selection:
  part 구조체의 멤버에 접근하려면 배열 첨자와 멤버 선택을 해야함

```
inventory[i].number = 883;
```

- Accessing a single character in a part name requires subscripting, followed by selection, followed by subscripting:
  part 이름의 한 글자를 읽을 때는 배열 요소 선택용 첨자, 구조체의 멤버 선택, 멤버 배열의 첨자 선택을 함

```
inventory[i].name[0] = '\0';
```

Initializing an Array of Structures

- Initializing an array of structures is done in much the same way as initializing a multidimensional array.
  구조체 배열의 초기화는 다차원 배열 초기화와 유사한 방식으로 가능
- Each structure has its own brace-enclosed initializer; the array initializer wraps another set of braces around the structure initializers.
  각 구조체는 중괄호로 묶어서 초기화함; 중괄호가 중첩된 구조임 구조체 초기화 중괄호를 배열 초기화 중괄호가 둘서 쌈

Initializing an Array of Structures

- One reason for initializing an array of structures is that it contains information that won't change during program execution.
  배열로 구조체를 초기화하는 이유는 프로그램 실행되는 동안에 해당 정보가 변경되지 않을 정보이기 때문
- Example: an array that contains country codes used when making international telephone calls. 예: 각 나라 별 국제 전화번호 코드
- The elements of the array will be structures that store the name of a country along with its code:
  선언된 구조체 배열은 나라 이름과 코드 번호를 저장함

```
struct dialing_code {
  char *country;
  int code;
};
```

Initializing an Array of Structures

```
const struct dialing_code country_codes[] =
  {{"Argentina",             54}, {"Bangladesh",       880},
   {"Brazil",                55}, {"Burma (Myanmar)",   95},
   {"China",                 86}, {"Colombia",          57},
   {"Congo, Dem. Rep. of", 243}, {"Egypt",             20},
   {"Ethiopia",            251}, {"France",            33},
   {"Germany",              49}, {"India",             91},
   {"Indonesia",            62}, {"Iran",              98},
   {"Italy",                39}, {"Japan",             81},
   {"Mexico",               52}, {"Nigeria",          234},
   {"Pakistan",             92}, {"Philippines",       63},
   {"Poland",               48}, {"Russia",             7},
   {"South Africa",         27}, {"South Korea",       82},
   {"Spain",                34}, {"Sudan",            249},
   {"Thailand",             66}, {"Turkey",            90},
   {"Ukraine",             380}, {"United Kingdom",    44},
   {"United States",         1}, {"Vietnam",           84}};
```

- The inner braces around each structure value are optional.
  각 구조체 초기값의 중괄호도는 선택사항임

Initializing an Array of Structures

- C99's designated initializers allow an item to have more than one designator.
  c99의 지정초기화는 하나 이상의 지정자를 갖을 수 있음
- A declaration of the `inventory` array that uses a designated initializer to create a single part:
  지정 초기화로 하나의 부품만 inventory 배열 구조체의 값을 초기화하는 예

```
struct part inventory[100] =
  {[0].number = 528, [0].on_hand = 10,
   [0].name[0] = '\0'};
```

The first two items in the initializer use two designators; the last item uses three.
지정자로 첨자 0의 구조체의 멤버를 초기화 함, 순서는 지정하였기 때문에 원 구조체 정의와 다름

Program: Maintaining a Parts Database

- The `inventory.c` program illustrates how nested arrays and structures are used in practice.
  inventory.c 프로그램은 배열과 구조체가 중첩된 경우의 활용 예를 보임
- The program tracks parts stored in a warehouse.
  프로그램은 창고의 부품들을 관리함
- Information about the parts is stored in an array of structures.
  부품의 정보는 구조체 배열에 저장됨
- Contents of each structure: 각 구조체의 내용은 다음과 같음
  - Part number
  - Name
  - Quantity

Program: Maintaining a Parts Database

- Operations supported by the program:
  프로그램이 지원하는 동작
  - Add a new part number, part name, and initial quantity on hand 새로운 부품 번호, 이름, 개수를 추가
  - Given a part number, print the name of the part and the current quantity on hand 부품 번호에 대한 이름과 보유 개수를 출력
  - Given a part number, change the quantity on hand
    부품 번호에 대해 보유 개수 수정
  - Print a table showing all information in the database
    데이터베이스의 모든 정보 출력
  - Terminate program execution 프로그램 종료

Program: Maintaining a Parts Database

- The codes `i` (insert), `s` (search), `u` (update), `p` (print), and `q` (quit) will be used to represent these operations.
  각 기능은 i (삽입), s (검색), u (갱신), p (출력) q (종료)로 조작 가능
- A session with the program: 실행 예

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```

Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```

Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number   Part Name                 Quantity on Hand
    528       Disk drive                       8
    914       Printer cable                    5

Enter operation code: q
```

Program: Maintaining a Parts Database

- The program will store information about each part in a structure.
  각 부품 정보를 구조체에 저장
- The structures will be stored in an array named `inventory`.
  inventory라는 구조체 배열로 정보 취합
- A variable named `num_parts` will keep track of the number of parts currently stored in the array.
  num_parts 변수가 배열에 저장된 부품의 개수를 관리

Program: Maintaining a Parts Database

- An outline of the program's main loop: 프로그램의 메인 루프

```
for (;;) {
   prompt user to enter operation code; //동작 코드
   read code;  // 코드 읽기
   switch (code) {
      case 'i': perform insert operation; break;// 삽입
      case 's': perform search operation; break;// 검색
      case 'u': perform update operation; break;// 갱신
      case 'p': perform print operation; break;// 출력
      case 'q': terminate program;// 종료
      default:  print error message;// 에러 출력
   }
}
```

Program: Maintaining a Parts Database

- Separate functions will perform the insert, search, update, and print operations.
  각 동작에 맞게 새로운 함수 정의
- Since the functions will all need access to `inventory` and `num_parts`, these variables will be external.
  inventory와 num_parts는 모든 함수에서 활용해야 함; external로 선언
- The program is split into three files: 3 부분으로 구성함
  - `inventory.c` (the bulk of the program) 프로그램의 메인 코드
  - `readline.h` (contains the prototype for the `read_line` function) read_line함수의 프로토타입/원형이 있음
  - `readline.c` (contains the definition of `read_line`)
    read_line의 구현/정의가 됨

inventory.c

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0;   /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

```
/**************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **************************************************/
int main(void)
{
  char code;
  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
```

```
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```

```
/*************************************************
 * find_part: Looks up a part number in the inventory     *
 *            array. Returns the array index if the part  *
 *            number is found; otherwise, returns -1.     *
 *************************************************/
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}
```

```
/**************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         database. Prints an error message and returns  *
 *         prematurely if the part already exists or the  *
 *         database is full.                              *
 **************************************************/
void insert(void)
{
  int part_number;

  if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }
```

```
  printf("Enter part number: ");
  scanf("%d", &part_number);
  if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
  }

  inventory[num_parts].number = part_number;
  printf("Enter part name: ");
  read_line(inventory[num_parts].name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &inventory[num_parts].on_hand);
  num_parts++;
}
```

```
/**********************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************/
void search(void)
{
  int i, number;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Part name: %s\n", inventory[i].name);
    printf("Quantity on hand: %d\n", inventory[i].on_hand);
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int i, number, change;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    inventory[i].on_hand += change;
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Parts are printed in the      *
 *        order in which they were entered into the       *
 *        database.                                       *
 **********************************************/
void print(void)
{
  int i;

  printf("Part Number   Part Name                    "
         "Quantity on Hand\n");
  for (i = 0; i < num_parts; i++)
    printf("%7d       %-25s%11d\n", inventory[i].number,
           inventory[i].name, inventory[i].on_hand);
}
```

Program: Maintaining a Parts Database

- The version of `read_line` in Chapter 13 won't work properly in the current program. 13장의 read_line은 정상 동작 안함
- Consider what happens when the user inserts a part:
  다음처럼 입력하는 경우의 예를 보자

```
Enter part number: 528
Enter part name: Disk drive
```

- The user presses the Enter key after entering the part number, leaving an invisible new-line character that the program must read.
  part number를 입력할 때 숫자 외에 눈에 안보이는 줄바꿈 기호(엔터)도 입력함
- When `scanf` reads the part number, it consumes the 5, 2, and 8, but leaves the new-line character unread.
  scanf가 읽을 때 5, 2, 8을 읽지만 줄바꿈 기호는 읽지 않음

Program: Maintaining a Parts Database

- If we try to read the part name using the original `read_line` function, it will encounter the new-line character immediately and stop reading.
  원래의 read_line 함수는 부품 이름을 읽을 때 줄바꿈 기호를 문자열로 인식하는 문제가 있음; 결과적으로 아무것도 입력하지 않았지만, 더 이상 읽지 않음
- This problem is common when numerical input is followed by character input. 숫자 입력 후 글자 입력하는 경우 흔히 발생하는 문제임
- One solution is to write a version of `read_line` that skips white-space characters before it begins storing characters. 해결 방법 중 하나는 글자를 읽을 때는 공백 글자 무시하도록 read_line을 수정
- This solves the new-line problem and also allows us to avoid storing blanks that precede the part name.
  줄바꿈 및 이름앞의 공백을 제거하는 효과가 있음

readline.h

```
#ifndef READLINE_H
#define READLINE_H

/**************************************************
 * read_line: Skips leading white-space characters, then  *
 *            reads the remainder of the input line and   *
 *            stores it in str. Truncates the line if its *
 *            length exceeds n. Returns the number of     *
 *            characters stored.                          *
 **************************************************/
int read_line(char str[], int n);

#endif
```

readline.c

```
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;
  while (ch != '\n' && ch != EOF) {
    if (i < n)
      str[i++] = ch;
    ch = getchar();
  }
  str[i] = '\0';
  return i;
}
```

Unions

Unions

- A ***union,*** like a structure, consists of one or more members, possibly of different types.
  union도 구조체와 같이 서로 다른 형을 갖는 하나 이상의 멤버들로 구성
- The compiler allocates only enough space for the largest of the members, which overlay each other within this space.
  컴파일러는 가장 큰 멤버들의 크기를 보고 충분한 공간을 할당하고, 멤버를 그 공간 안에 저장함
- Assigning a new value to one member alters the values of the other members as well.
  한 멤버에 새로운 값을 할당하면 다른 멤들의 값도 영향을 받음

Unions

- An example of a union variable: 유니언 변수의 선언 예

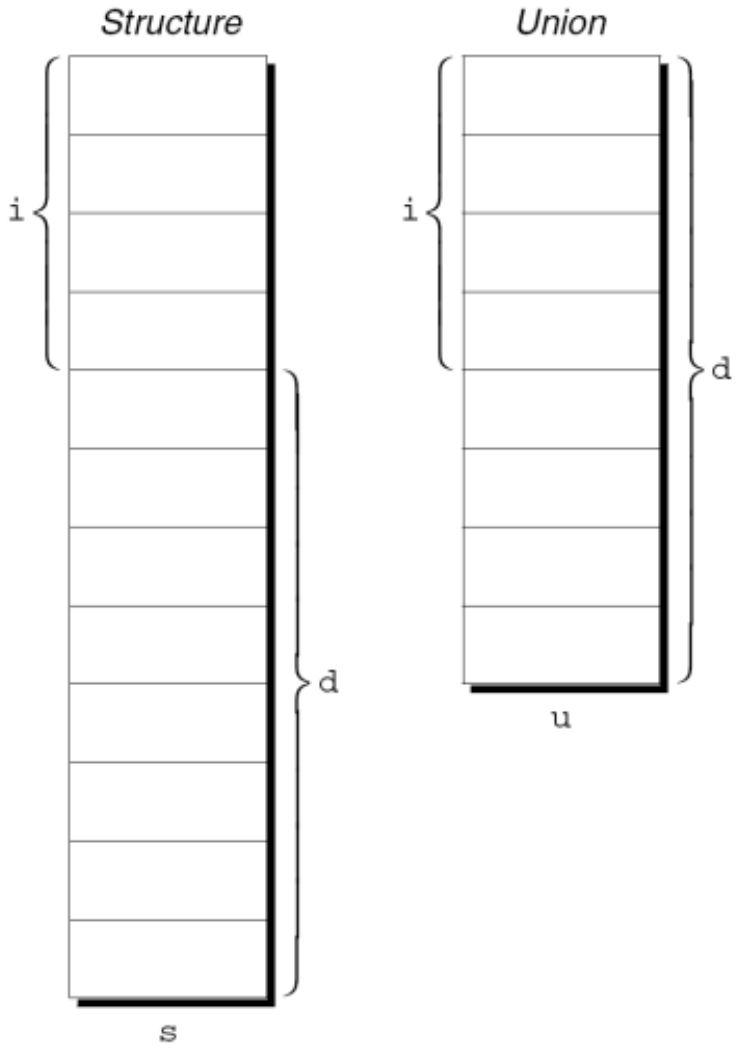
```
union {
  int i;
  double d;
} u;
```

- The declaration of a union closely resembles a structure declaration: 유니언 변수의 선언은 구조체의 선언과 유사함

```
struct {
  int i;
  double d;
} s;
```

Unions

- The structure `s` and the union `u` differ in just one way.
  구조체 s와 유니언 u는 하나의 차이만 존재함
- The members of `s` are stored at different addresses in memory.
  s의 멤버는 서로 다른 주소 공간에 저장됨
- The members of `u` are stored at the same address.
  u의 멤버는 같은 주소를 갖음



Unions

- Members of a union are accessed in the same way as members of a structure:
  유니언 멤버의 접근 방법은 구조체 멤버의 접근과 동일함

```
u.i = 82;
u.d = 74.8;
```

- Changing one member of a union alters any value previously stored in any of the other members.
  유니언의 한 멤버 변수를 저장하면 그 전에 저장되었던 멤버의 값에 영향을 줌
  - Storing a value in `u.d` causes any value previously stored in `u.i` to be lost.
    u.d에 값을 저장하면 u.i에 저장된 값은 소멸됨
  - Changing `u.i` corrupts `u.d`.
    u.i 저장하면 u.d에 저장된 값은 깨짐

Unions

- The properties of unions are almost identical to the properties of structures.
  유니언의 성질은 구조체의 성질과 대부분 같음
- We can declare union tags and union types in the same way we declare structure tags and types.
  유니언 태그나 유니언 형을 선언하여 쓸 수 있음
- Like structures, unions can be copied using the = operator, passed to functions, and returned by functions.
  유니언은 = 연산자로 복사되고 함수의 인자로 전달, 리턴 받을 수 있음

Unions

- Only the first member of a union can be given an initial value.
  단, 유니언의 첫 번째 멤버만 초기화될 수 있음
- How to initialize the `i` member of `u` to 0:
  u의 멤버 i를 0으로 초기화하는 예

```
union {
  int i;
  double d;
} u = {0};
```

- The expression inside the braces must be constant. (The rules are slightly different in C99.)
  중괄호 안의 표현식은 상수여야 함 (c99는 다른 규칙을 따름)

Unions

- Designated initializers can also be used with unions.
  union에서도 지정 초기화를 쓸 수 있음
- A designated initializer allows us to specify which member of a union should be initialized: 어떤 멤버를 초기화 할 지 지정 초기화로 정함

```
union {
  int i;
  double d;
} u = {.d = 10.0};
```

- Only one member can be initialized, but it doesn't have to be the first one. 단, 여러 멤버 중 하나의 값만 초기화 할 수 있음

Unions

- Applications for unions: 유니언의 응용처
  - Saving space 공간 절약
  - Building mixed data structures 혼합된 자료 구조 생성
  - Viewing storage in different ways (discussed in Chapter 20)
    저장장치를 보는 또 다른 시각 (20장에서 자세히 다룸)

Using Unions to Save Space

- Unions can be used to save space in structures.
  구조체의 공간을 줄이기 위해 유니언을 사용 가능
- Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog.
  예로, 선물카탈로그 책자의 아이템의 정보를 저장하는 구조체를 생각해보자
- Each item has a stock number and a price, as well as other information that depends on the type of the item:
  각 아이템은 품명, 금액, 그 외 관련 정보를 갖음

  Books: Title, author, number of pages 제목, 저자, 페이지 수

  Mugs: Design 디자인

  Shirts: Design, colors available, sizes available 디자인, 남은 색상, 남은 치수

Using Unions to Save Space

- A first attempt at designing the `catalog_item` structure: 첫 번째 catalog_item 구조체 작성 시도

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN+1];
  char author[AUTHOR_LEN+1];
  int num_pages;
  char design[DESIGN_LEN+1];
  int colors;
  int sizes;
};
```

Using Unions to Save Space

- The `item_type` member would have one of the values `BOOK`, `MUG`, or `SHIRT`. item_type멤버는 BOOK, MUG, SHIRT 중 하나만 값이 있음
- The `colors` and `sizes` members would store encoded combinations of colors and sizes.
  colors와 sizes 멤버는 색과 크기의 값을 저장
- This structure wastes space, since only part of the information in the structure is common to all items in the catalog.
  이 구조체는 공간 낭비가 심함; 몇 개의 정보만 모든 아이템에 공통으로 사용됨
- By putting a union inside the `catalog_item` structure, we can reduce the space required by the structure.
  catalog_item구조체에 유니언을 씀으로 공간을 절약할 수 있음

Using Unions to Save Space

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  union {
    struct {
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
    } mug;
    struct {
      char design[DESIGN_LEN+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

Using Unions to Save Space

- If `c` is a `catalog_item` structure that represents a book, we can print the book's title in the following way:
  c가 책을 나타내는 catalog_item 구조체라면 다음과 같이 나타낼 수 있음

```
printf("%s", c.item.book.title);
```

- As this example shows, accessing a union that's nested inside a structure can be awkward.
  구조체에 유니언이 있고 그 안에 또 구조체가 있는 중첩 구조를 접근하는 것이 이상해 보일 수 있음

Using Unions to Save Space

- The `catalog_item` structure can be used to illustrate an interesting aspect of unions.
  catalog_item구조체는 유니언의 재미있는 성질을 보여줌
- Normally, it's not a good idea to store a value into one member of a union and then access the data through a different member.
  일반적으로 한 유니언의 멤버를 저장하고 다른 멤버를 통해 값을 읽어오는 것은 현명하지 못함

Using Unions to Save Space

- However, there is a special case: two or more of the members of the union are structures, and the structures begin with one or more matching members.
  특수한 경우 제외: 2개 이상의 멤버가 구조체이고 각 구조체마다 동일한 멤버를 갖는 경우
- If one of the structures is currently valid, then the matching members in the other structures will also be valid.
  한 구조체가 유효하면 다른 구조체도 유효함

Using Unions to Save Space

- The union embedded in the `catalog_item` structure contains three structures as members.
  catalog_item구조체는 유니언으로 세 개의 구조체 멤버를 갖음
- Two of these (`mug` and `shirt`) begin with a matching member (`design`). mug와 shirt는 동일한 멤버(design)를 갖고 있음
- Now, suppose that we assign a value to one of the `design` members: design 멤버 중 하나에 값을 저장하는 예

```
strcpy(c.item.mug.design, "Cats");
```

- The `design` member in the other structure will be defined and have the same value:
  다른 구조체의 design 멤버를 접근해도 같은 값을 갖음

```
printf("%s", c.item.shirt.design);
  /* prints "Cats" */
```

Using Unions to Build Mixed Data Structures

- Unions can be used to create data structures that contain a mixture of data of different types.
  유니언은 서로 다른 데이터 형으로 구성된 자료 구조를 만드는데 쓰임
- Suppose that we need an array whose elements are a mixture of `int` and `double` values.
  어떤 배열이 int 또는 double 값을 갖는다고 해보자
- First, we define a union type whose members represent the different kinds of data to be stored in the array:
  서로 다른 형을 갖는 멤버로 구성된 유니언 형을 만들어 배열을 선언할 수 있음

```
typedef union {
  int i;
  double d;
} Number;
```

Using Unions to Build Mixed Data Structures

- Next, we create an array whose elements are `Number` values:
  유니언 형을 만든 후, 그 형을 이용해 배열을 선언함

```
Number number_array[1000];
```

- A `Number` union can store either an `int` value or a `double` value. 이제 Number 유니언 형으로 만든 배열은 int와 double 값을 갖을 수 있음
- This makes it possible to store a mixture of `int` and `double` values in `number_array`:
  number_array 배열에 정수 / 실수 값을 저장할 수 있음

```
number_array[0].i = 5;
number_array[1].d = 8.395;
```

Adding a "Tag Field" to a Union

- There's no easy way to tell which member of a union was last changed and therefore contains a meaningful value.
  유니언의 멤버 중 어떤 것이 의미있는 값을 갖고 있는지 파악하는 것이 쉽지 않음
- Consider the problem of writing a function that displays the value stored in a `Number` union:
  Number 유니언에 저장된 값을 출력하는 함수를 작성한다고 하자

```
void print_number(Number n)
{
  if (n contains an integer)
    printf("%d", n.i);
  else
    printf("%g", n.d);
}
```

There's no way for `print_number` to determine whether `n` contains an integer or a floating-point number.
print_number는 n이 정수인지 실수인지 판단할 수 없음

Adding a “Tag Field” to a Union

- In order to keep track of this information, we can embed the union within a structure that has one other member: a “tag field” or “discriminant.”
  구조체 내에 이 정보를 파악할 수 있도록 구분자 또는 태그 정보를 포함 할 수 있음
- The purpose of a tag field is to remind us what’s currently stored in the union.
  태그 필드는 유니언에 무엇이 저장되었는지 알려줌
- `item_type` served this purpose in the `catalog_item` structure.
  catalog_item구조체에서는 item_type이 그 역할을 함

Adding a "Tag Field" to a Union

- The `Number` type as a structure with an embedded union:
  Number 형을 구조체로 선언하고 그 안에 유니언을 포함시킴

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
  int kind;    /* tag field */
  union {
    int i;
    double d;
  } u;
} Number;
```

- The value of `kind` will be either `INT_KIND` or `DOUBLE_KIND`.
  kind의 값에 INT_KIND 또는 DOUBLE_KIND로 표시함

Adding a "Tag Field" to a Union

- Each time we assign a value to a member of `u`, we'll also change `kind` to remind us which member of `u` we modified.
  u의 멤버에 값을 할당할 때마다 kind의 값을 갱신해야함
- An example that assigns a value to the `i` member of `u`:
  u의 멤버로 i의 값을 바꾸는 예

```
n.kind = INT_KIND;
n.u.i = 82;
```

`n` is assumed to be a `Number` variable.
n은 Number 형 변수라 가정

Adding a “Tag Field” to a Union

- When the number stored in a `Number` variable is retrieved, `kind` will tell us which member of the union was the last to be assigned a value.
  Number 형 변수 kind의 값으로 어떤 유니언 값이 마지막으로 쓰였는지 알수 있음
- A function that takes advantage of this capability:
  그 정보를 활용하는 함수의 예

```
void print_number(Number n)
{
  if (n.kind == INT_KIND)
    printf("%d", n.u.i);
  else
    printf("%g", n.u.d);
}
```

Enumerations

Enumerations

- In many programs, we'll need variables that have only a small set of meaningful values.
  어떤 프로그램들은 의미있는 값의 범위가 한정적일 수 있음
- A variable that stores the suit of a playing card should have only four potential values: "clubs," "diamonds," "hearts," and "spades."
  예: 카드 게임에서는 "클럽", "다이어몬드", "하트", 그리고 "스페이드"만 있음

Enumerations

- A “suit” variable can be declared as an integer, with a set of codes that represent the possible values of the variable:
  “suit”라는 변수는 정수를 저장하고 패와 수를 지정하여 구분할 수 있음

```
int s;    /* s will store a suit */
…
s = 2;    /* 2 represents "hearts" */
```

- Problems with this technique:
  이와 같은 접근 방식에 문제점
  - We can’t tell that `s` has only four possible values.
    s의 값으로 4개 이외의 값만 있는지 확신할 수 없음
  - The significance of 2 isn’t apparent.
    2의 의미가 명시적이지 않음

Enumerations

- Using macros to define a suit “type” and names for the various suits is a step in the right direction:
  매크로 정의를 통해 “type”과 이름을 정하는 것은 괜찮은 방법임

```
#define SUIT     int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS   2
#define SPADES   3
```

- An updated version of the previous example: 앞 선 예의 개선된 버전

```
SUIT s;
…
s = HEARTS;
```

Enumerations

- Problems with this technique:
  이 방법의 문제점
  - There's no indication to someone reading the program that the macros represent values of the same "type."
    매크로의 정의가 같은 "type"의 그룹 내의 값인지 파악하기 어려움
  - If the number of possible values is more than a few, defining a separate macro for each will be tedious.
    그룹 내에 값이 한 두개가 아니라면 매크로 정의를 작성하는 것도 만만치 않음
  - The names `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` will be removed by the preprocessor, so they won't be available during debugging.
    전처리 과정에서 매크로 정의는 사라지기 때문에 컴파일 이후 디버깅 과정에서 정보를 활용할 수 없음

Enumerations

- C provides a special kind of type designed specifically for variables that have a small number of possible values.
  값의 범위가 한정되어 있는 변수 용 타입을 제공함
- An ***enumerated type*** is a type whose values are listed ("enumerated") by the programmer. enumerated type(열거 형)이라 부르고 값은 프로그래머에 의해 나열됨
- Each value must have a name (an ***enumeration constant***).
  각 값은 이름을 갖음

Enumerations

- Although enumerations have little in common with structures and unions, they're declared in a similar way:
  열거 형이 유니언과 구조체와 유사한 점은 없지만, 선언 방식은 유사함

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

- The names of enumeration constants must be different from other identifiers declared in the enclosing scope.
  열거 형에 사용된 상수의 이름은 범위 내에서 유일해야 함

Enumerations

- Enumeration constants are similar to constants created with the `#define` directive, but they're not equivalent.
  #define으로 선언된 상수와 비슷하지만 동일하지는 않음
- If an enumeration is declared inside a function, its constants won't be visible outside the function.
  열거 형이 함수 내에서 선언되었다면 해당 함수 밖에서 보이지 않음

Enumeration Tags and Type Names

- As with structures and unions, there are two ways to name an enumeration: by declaring a tag or by using `typedef` to create a genuine type name.
  구조체와 유니언과 동일하게 태그와 typedef으로 새로운 형의 이름을 만들 수 있음
- Enumeration tags resemble structure and union tags:
  열거 형 태그는 유니언과 구조체의 태그와 같은 방식으로 선언

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

- `suit` variables would be declared in the following way:
  suit 열거 형 변수는 다음과 같이 선언

```
enum suit s1, s2;
```

Enumeration Tags and Type Names

- As an alternative, we could use `typedef` to make `Suit` a type name:
  typedef으로 Suit이라는 이름의 새로운 형을 만들 수도 있음

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

- In C89, using `typedef` to name an enumeration is an excellent way to create a Boolean type:
  c89에서는 불리언 형을 만드는 가장 좋은 방법은 typedef 을 쓰는 것임

```
typedef enum {FALSE, TRUE} Bool;
```

Enumerations as Integers

- Behind the scenes, C treats enumeration variables and constants as integers.
  C에서는 열거 형 변수와 상수를 정수로 판단함
- By default, the compiler assigns the integers 0, 1, 2, … to the constants in a particular enumeration.
  열거 형에 나열된 순서에 따라 0, 1, 2, …의 값을 할당 받음
- In the `suit` enumeration, `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` represent 0, 1, 2, and 3, respectively.
  suit 열거 형에서는 클럽, 다이어몬드, 하트, 스페이드가 각각 0, 1, 2, 3의 값을 갖음

Enumerations as Integers

- The programmer can choose different values for enumeration constants: 프로그래머는 열거되는 상수에 다른 값을 지정할 수 있음

```
enum suit {CLUBS = 1, DIAMONDS = 2,
           HEARTS = 3, SPADES = 4};
```

- The values of enumeration constants may be arbitrary integers, listed in no particular order: 상수의 값은 순서와 상관없음

```
enum dept {RESEARCH = 20,
           PRODUCTION = 10, SALES = 25};
```

- It's even legal for two or more enumeration constants to have the same value. 두 개의 열거 값이 같은 값을 가져도 됨

Enumerations as Integers

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant.
  어떤 열거 값이 값을 지정받지 못했다면, 이전 상수 값보다 1 큰 값을 받음
- The first enumeration constant has the value 0 by default.
  첫 번째 열거 상수 값은 기본적으로 0을 갖음
- Example:

```
enum EGA_colors {BLACK, LT_GRAY = 7,
                 DK_GRAY, WHITE = 15};
```

`BLACK` has the value 0, `LT_GRAY` is 7, `DK_GRAY` is 8, and `WHITE` is 15.
BLACK은 0, LT_GRAY는 7, DK_GRAY는 8, WHITE는 15임

Enumerations as Integers

- Enumeration values can be mixed with ordinary integers:
  열거 값은 다른 정수들과 혼합되어 쓰일 수 있음

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;

i = DIAMONDS;    /* i is now 1              */
s = 0;           /* s is now 0 (CLUBS)      */
s++;             /* s is now 1 (DIAMONDS)   */
i = s + 2;       /* i is now 3              */
```

- `s` is treated as a variable of some integer type.
  s는 정수 타입의 변수처럼 취급됨
- `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` are names for the integers 0, 1, 2, and 3.
  클럽, 다이어몬드, 하트, 스페이드는 0, 1, 2, 3의 값을 갖고 있음

Enumerations as Integers

- Although it's convenient to be able to use an enumeration value as an integer, it's dangerous to use an integer as an enumeration value.
  열거 형 변수가 정수형이더라도, 정수 형 변수로 취급하여 값을 저장하면 안됨
- For example, we might accidentally store the number 4—which doesn't correspond to any suit—into `s`.
  열거 형 변수의 범위 밖의 값을 저장하게 되면 무의미한 값을 저장하게 됨

Using Enumerations to Declare "Tag Fields"

- Enumerations are perfect for determining which member of a union was the last to be assigned a value.
  열거 형 변수는 유니언 중 어떤 멤버가 사용되었는지 표시하기 좋음
- In the `Number` structure, we can make the `kind` member an enumeration instead of an `int`:
  Number 구조체에서 kind를 열거 형 변수로 선언한 예

```
typedef struct {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int i;
    double d;
  } u;
} Number;
```

Using Enumerations to Declare "Tag Fields"

- The new structure is used in exactly the same way as the old one.
  새로운 구조체는 과거에 정의된 구조체와 동일한 기능을 함
- Advantages of the new structure:
  새로운 구조체의 장점
  - Does away with the `INT_KIND` and `DOUBLE_KIND` macros
    매크로 정의 없이 선언 됨
  - Makes it obvious that `kind` has only two possible values:
    `INT_KIND` and `DOUBLE_KIND`
    두 개의 유일한 값의 범위를 갖음