Structures

Chapter 12

- Collection of variables referenced with one name
- A structure declaration is a blueprint, or "cookie cutter" (i.e., does not allocate memory); often put in header files
- struct tag {
  variable declarations;
};
- Taken together, struct tag creates a new type specifier
- Variable declarations may include any type except the struct-type being created
- Each element or variable is called a member or field
- The tag can be any legal identifier

struct Examples

- struct variables and struct pointers

```c
struct date {
  int year;
  int mon;
  int day;
};

struct date when;
struct date* datePtr;

Space is not allocated by struct declarations, only by variable definitions
struct declarations often go in header files
```

Member Access Operators

- Accessing the individual data contained in a struct

```c
If the LHS is a struct object, use the . (the period, pronounced "dot")
If the LHS is a pointer to a struct, use the -> (formed by the minus and greater than symbols, pronounced "arrow")
Members are legal anywhere a variable of that type is legal (i.e., they can be either l-values or r-values)
Members are in a unique scope
Examples
  * when.year = 1998;
  * int mon = when.mon;
  * datePtr->day = 14;
  * int year = datePtr->year;
```

struct Facts

- Assignment, pass, and return copies a "patch of bits"

```c
Initialization
struct date today = { 1998, 2, 14 };
Assignment (bit-by-bit copy)
when = today;
Member by member assignment (if you only want some members)
when.year = today.year;
when.mon = today.mon;
when.day = today.day;
Function arguments are passed by value (bit-by-bit copy)
Functional return values are returned by value (bit-by-bit copy)
```

Manipulating structs

- Miscellaneous syntax

```c
Finding the address: &today;
Finding the size (in bytes): sizeof(struct date);
Allocating structs dynamically:
  struct date* dateVar;
  dateVar = (struct date*) malloc(sizeof(struct date));
Deallocationg dynamic memory or variables: free(dateVar);
A common error: memory overflow
  * struct date* dPointer;
  dPointer->year = 1998;
  fread(dPointer, sizeof(struct date), 1, fp);
```
Structure Function Arguments

Passing structs by value and by "reference"

```c
void printDate1(struct date D)
{
    printf("%s\n", D.year);
    printf("%s\n", D.month);
    printf("%s\n", D.day);
}

struct date newYears = {1, 1, 1998};
printDate1(newYears);
```

```c
void printDate2(struct date* D)
{
    printf("%s\n", D->year);
    printf("%s\n", D->month);
    printf("%s\n", D->day);
}

struct date newYears = {1, 1, 1998};
printDate2(&newYears);
```

Structure Return Values

Return by value

```c
struct date getDate(void)
{
    struct date D;
    printf("enter a year: ");
    scanf("%d", &D.year);
    printf("enter a month: ");
    scanf("%d", &D.month);
    printf("enter a day: ");
    scanf("%d", &D.day);
    return D;
}
```

Arrays and structs

```c
struct containing arrays and arrays of structs
```

```c
struct containing arrays
{
    char year[5];
    char mon[4];
    char day[3];
}
```

```c
structs containing arrays
```

```c
struct date Important[100];
strcpy(Important[5].year, "1776");
strcpy(Important[5].month, "Jul");
strcpy(Important[5].day, "4");
if (Important[5].year[0] == '1')
...
```

Arrays of structs

```c
struct data Holiday
{
    char* year;
    char* mon;
    char* day;
};
```

```c
struct date Holiday;
struct date temp;
Holiday.year = "1847",
temp = Holiday;
temp = strdup(Holiday.year);
```

Nested structs

```c
struct within a struct
```

```c
struct range {
    struct date Start;
    struct date End;
};
```

```c
struct date {
    int year, mon, day;
    struct date end; /*error*/
};
```

```c
struct ListNode {
    char* Name;
    struct ListNode* Next;
};
```

Alternate Declarations and Definitions

Alternate syntax

- Create a type and define variables:
  ```c
  struct tag {
      variable declarations;
  } var1, var2;
  ```

- Define variables only:
  ```c
  struct {
      variable declarations;
  } var1, var2;
  ```

- Create a type that doesn't require the struct keyword:
  ```c
  typedef struct {
      variable declarations;
  } short_name;
  ```
**unionS**

A "schizophrenic" type

- Look like structs but use the keyword `union`
- Can contain data of multiple types but only one type at a time
  - structs reserve memory for all fields
  - unions reserve memory for the largest field (memory is shared)
  - structs and unions are often used together
- May be used to create variables capable of storing more than one type of data
- Fields are accessed with either . (dot) or -> (arrow)
- May be used for type conversions (but may not be portable)
  - data stored in one union element (as a bit pattern)
  - data read out of another union element (as a bit pattern)

---

**struct VS union**

Memory layout and consumption

```
struct demo {
    char c;
    int i;
    double d;
    char* s;
};
```

```
union constant {
    char c;
    int i;
    double d;
    char* s;
};
```

---

**struct / union Example**

Easy, portable, type-independent data manipulation

```
enum types { Char, Int, Double, String };
void print(struct values data)
{
    switch(data.type)
    {
        case Char:
            printf("%c", data.value.c);
            break;
        case Int:
            printf("%d", data.value.i);
            break;
        case Double:
            printf("%g", data.value.d);
            break;
        case String:
            printf("%s", data.value.s);
            break;
    }
}
```

---

**Bit Fields**

Small, odd-sized data

- Based on structs (i.e., are an extension of the struct syntax)
- Used to access non-standard sized data (i.e., not 1, 2, 4, or 8 bytes); usually less than 1 byte (8 bits)
- Each field is followed by a size, measured in bits
- Example:

```
struct tag {
    unsigned int field1 : 3;
    unsigned int field2 : 7;
    unsigned int field3 : 6;
};
```

---

**Unix Bit Field Example**

From `getsnodes.c`

```
struct modes {
    unsigned int pad : 16;
    unsigned int type : 7;
    unsigned int user : 3;
    unsigned int group : 3;
    unsigned int others: 3;
};
```

```
union convert {
    mode_t file_mode;
    struct modes split;
};
```

---

**DOS/Windows Bit Field Example**

From `fstamp.c`

```
struct file_date {
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year : 7;
};
```

```
union toDate {
    int reg;
    struct file_date date;
};
```