Chapter 7: Structuring the Data
Topics

• Introduction to Structures
• Operations on Structures
• Using Structures with Arrays
• Using Structures with Pointers
• Passing Structures to and from Functions
• Bit Fields
• Enumerated Types
• Unions
Structures

- A **structure** is a data type that can hold multiple fields of multiple types of data.
- A structure is not an array, an array holds only a table of the same type.
- A structure is a user defined data type. Instead of float, int, or char, the user creates a type of data.
- The keyword for this construction is `struct`.
struct tag
{
    member_type member_name;
    member_type member_name;
    member_type member_name;
};
Structures

• Why structures?
  • Better to describe the real world
    • The world is more than just one set of numbers
    • The world is more than just one kind of data
  • Good to hold associated data
    • Address books
    • phone books
    • calendars
  • Now a method of keeping a record is available, new methods can be applied
struct book
{
    char title[1024];
    char author[1024];
    double dewy;
    long pages;
    int year;
};

struct book book1;
Struct

- Each portion of a struct is called a field.
- To access the field of a struct, a period is used.
- By doing this, each field can be seen by the statements of the program.
Struct

• To create a variable using this structure, the struct can be used later on.

• Structure can be created without a name tag for the struct, but will not be able to be used for declaring variables.
struct
{
    char title[1024];
    char author[1024];
    double dewy;
    long pages;
    int year;
} book1, book2;
struct book WarAndPeace;

- Will contain nonsense
- Fields must be filled separately
struct book Almanac = {
  "2006 Almanac",
  "Richard Poorly",
  1022.322,
  365,
  2006
};

- Each data field is the right type
- Each data field is in the right order
- Will be correct
struct book Encyclopedia={"Americana 2003"};

• Will only have some fields correct.
• Uninitialized arrays and strings may contain nonsense
• Numeric fields will contain zeros
Struct

- Each field of a struct is called a member
- Each member can be accessed with a period.
- `variable_name.member_name;`
Struct

```c
void ShowBook(struct book thisbook)
{
    printf("Title:\t%s\n",thisbook.title);
    printf("Author:\t%s\n",thisbook.author);
    printf("Year:\t%04d\n",thisbook.year);
    printf("Dewy:\t%f\n",thisbook.dewy);
    printf("Pages:\t%d\n",thisbook.pages);
    printf("\n");
}
```
struct book DayBook;
strcpy(DayBook.title,"History");
strcpy(DayBook.author,"Heroditous");
DayBook.year=15;
DayBook.dewy=0.001;
DayBook.pages=2200;
Operations on Structures

• The period ‘.’ to access a member.
• The sizeof() command to find out the size.
• The address operator &, can be used to find the address of a struct.
• The assignment operator,’=‘ assigns the contents of one structure to another.
Using Structs with Arrays

• An array can be made out of any data type, including a user defined struct.

struct book booklist[5];
Using Structs with Arrays

- To reference a single cell, then the subscript or index is referenced.

booklist[3].title
booklist[3].author[5];
Using Structs with Arrays

- A nested initialization may be used with an array of structs

```c
    {"C by Discovery","Foster",1001.22,776,2005},
    {"Calculus","Edwards",1401.22,976,1997}
};
```
Using Structs with Pointers

- A pointer can be declared to reference any data type.

```c
struct book * bookptr=NULL;
```
Using Structs with Pointers

- A structure can not contain a copy of itself.
- THIS NEVER WORKS

```c
struct node
{
    int Data;
    struct node next;
};
```
Using Structs with Pointers

- But a struct can contain an address field. So a structure can contain a pointer to another structure of the same type.

```c
struct node
{
    int Data;
    struct node *next;
};
```
Using Structs with Pointers

• Assigning an address to a pointer is the same as before.

```c
struct book book1;
struct book * bookptr=NULL;
bookptr=&book1;
```
Using Structs with Pointers

- There are two methods of getting the information out of a member of a pointer to a struct.
- The first is the standard dereference operation *

\((\ast \text{bookptr})\).author
Using Structs with Pointers

• The second is the member selection operation, which is a minus – followed by a greater than. “->”

bookptr->author

• member selection is a more commonly used notation for referencing the member of a structure pointer.
Passing Structures to and from Functions

- Pointers to structures make them usable to pass to functions.
- By using pass by reference, the record can be sent to a function, and be processed using the member selection operator.
Passing Structures to and from Functions

```c
void Example2()
{
    struct book textbook;
    MakeBook(&textbook,"Embedded C","Micheal Pont",2002,380,123.456);
    ShowBook(textbook);
}

void MakeBook(struct book *thisbook,char *title,char *author, int year, long pages, double dewy)
{
    strcpy(thisbook->title, title);
    strcpy(thisbook->author, author);
    thisbook->year=year;
    thisbook->pages=pages;
    thisbook->dewy=dewy;
}
```
Bit Fields

- A *bit field* is a feature in C that allows a programmer to assign the contents of a struct to a specific number of bits.
- This is a technique used in situations when memory is limited on the computer.

```c
struct bitset
{
    unsigned int front: 3;
    unsigned int back: 5;
};
```
Enumerated types

- An **enumerated type** is a representation of a series of numbers in C using identifier names.

```c
enum tag
{
    identifier_1, identifier_2, ..., identifier_n
};
```

- This results in a list of identifiers numbered starting from 0 that can be used in place of integers.
Enumerated Types

- An enumerated type can also have a unique integer value explicitly set to the identifier.

```c
enum tag
{
  identifier_1=5, identifier_2=3, ..., identifier_n=22
};
```
Enumerator types

• A declaration of
enum boolean {true = 1, false = 0};

• would be similar to
#define false 0
#define true 1
Enumerated types

- `enum days {sun, mon, tue, wed, thu, fri, sat}`
- `#define sun 0`
- `#define mon 1`
- `#define tue 2`
- `#define wed 3`
- `#define thu 4`
- `#define fri 5`
- `#define sat 6`
Enumerated Types

• For example, a notation for the days of the week could be:

```c
enum days{ sun, mon, tue, wed, thu, fri, sat};
```

• with mon equal to 0 at the beginning of the count.

• Another example could be the months of summer with an explicit assignment

```c
enum summer={jun=6, jul=7, aug=8};
```
Enumerated Types

- Enumerated Types are also handy for working a switch statement.

```c
switch(today)
{
    case mon:
        printf("Monday\n");
        break;
    case tue:
        printf("Tuesday\n");
        break;
    case wed:
        printf("Wednesday\n");
        break;
    case thu:
        printf("Thursday\n");
        break;
    case fri:
        printf("Friday\n");
        break;
    case sat:
    case sun:
        printf("Weekend\n");
        break;
}
```
Enumerated Types

- Enumerated Types can also be used for handling the indexes of the subscripts of an array.

```c
int array[7]={15,16,17,18,19,20,21};
printf("Sunday  %d\n",array[sun]);
printf("Monday  %d\n",array[mon]);
printf("Tuesday %d\n",array[tue]);
printf("Wednesday%d\n",array[wed]);
printf("Thursday%d\n",array[thu]);
printf("Friday  %d\n",array[fri]);
printf("Saturday%d\n",array[sat]);
```
Enumerated Types

• Any integer operation, parameter, or other usage can also have an enumerated type in the same usage.
Unions

- A union is another type of structure that can hold many kinds of data.
- Unlike Struct, however, a union can only hold one piece of data at a time.

```c
union tag {
    member_type member_name;
    member_type member_name;
}
```
Unions

• Like struct, the tag is optional for declaring a union, but without a tag variables cannot be created later in the code.
Unions

- This union `speedrate` contains two different member variables of data.
- Only one member field of data can be occupied at a time.

```c
union speedrate {
    int miles_per_hour;
    float km_per_hour;
}
```
**Unions**

- Unions are accessed and operated upon just like structures.
- A `.` operator can be used to get access to a member variable.
- A `=` operator can be used to assign similar unions.
- A `sizeof()` operation can be used to find the size of the union.
- A `&` operation can be used to find the address of a union.
Unions

• Like Structures, Unions can be created as a pointer.
• As a pointer, the Union can be dereferenced with a “*”
• As a pointer, the member variables of the union can be manipulated using a member selection operator “->”
Unions

- The main function of a union is to reduce the amount of memory used in a program.
- A struct is the size of its entire member variable list.
- A union is the size of its largest member variable.