Structures

Non-elementary data type

- Also known as *aggregates*
- Allow the collection of different types of variables into one object
- Compared to arrays
  - In an array, all the elements are of the same type and are indexed
  - In a structure, each element (field) is named and has its own data type

Structure definitions

- Each structure is equivalent to a *record*, with each element of the structure being a *field* in the record
- Derived data types
  - Can be constructed by using the objects of other *known* data types
- Structure to represent information about a person

Name	$\leq$ 30 characters
Year of birth	integer
Height	integer

- Above can be declared in C as

```
struct person_info
{
    char name[31];
    int  yr_of_birth;
    int  height;        /* in inches */
}
people[100];
```

- `person_info` is the *tag* or symbolic name for the structure and can be omitted
  - Only the variables declared at the end of the structure declaration, (`people` in the above case), can have that structure type
- `people[100]` is the variable of type `person_info` and may be omitted as well
- Preferred way of declaration
  - Declare the structure type in the header file (the .h file) as

```
struct person_info
{
    char name[31];
    int  yr_of_birth;
    int  height;        /* in inches */
};
```

  - Declare the variable as

    ```
    struct person_info people[100];
    ```

  - The keyword `struct` must not be omitted

- The structure can be defined as a type obviating the need for using the keyword `struct` as follows:

  ```
  typedef struct
  {
      char name[31];
      int  yr_of_birth;
      int  height;        /* in inches */
  } person_info_t;
  ```

  The variable can now be declared as

  ```
  person_info_t people[100];
  ```

- Structures may not be compared
  - Structure members may not be necessarily stored in consecutive bytes
  - There also may be "holes" in a structure because computers may store specific data types only on certain memory boundaries, such as halfword, word, or doubleword boundaries
  - Example

    ```
    struct example
    {
        char c;
        int i;
    } sample1, sample2;
    ```

    on a 2-byte word machine, may get allocated as

Byte	Contents
0	01100001
1	Unused
2	00000000
3	01100001

Initializing structures

- The variables can be initialized if their memory space is permanent (external or static)

  ```
  static person_info_t
      leader = {"Bill Clinton", 1948, 73},
      president,
      people[100] = {
              {"Joe Smith",   1952, 70},
              {"John Doe",    1961, 65}
          };
  ```

- If the initializer is shorter than the structure being initialized, the remainder structure is filled with 0s
- The entire contents of a structure can be copied from one variable to another by using the simple assignment statement

  ```
  president = leader;
  ```

Accessing members of structures

- The individual elements in the structure can be referred to as *variable.fieldname* (by using the *dot operator*)

  ```
  president.name
  people[i].height
  people[i].name[j]
  ```

  The last one is to be read as `(people[i].name)[j]`
- Pointers and structures
  - Consider the following declaration

    ```
    person_info_t * ptr, people[100];
    ```
  - The statement

    ```
    ptr = people[i];
    ```

    allows us to have access to the ith element in the array as `*ptr`
  - An individual field in the element can be then referred as `(*ptr).height`
    - The parentheses around the pointer are required because `.` has a higher precedence than `*`
  - C allows us to refer to such fields by using the `->` operator as `ptr->height` (using the *arrow operator*)
    - `->` is known as *dereference and access member* operator
  - The individual characters in the `name` field can be referred to as `ptr->name[j]`

Functions and structures

- Structures may be passed to functions by either of the following mechanisms
  - Passing individual structure members
  - Passing an entire structure
  - Passing a pointer to a structure
- The first two mechanisms are used to pass the structure using call-by-value
  - Members of a caller's structure cannot be modified by the called function
- Arrays of structures, like all other arrays, automatically get passed by reference
- Structures can be used to pass arrays by value

  - Create a structure with the array as a member
  - Pass this structure by value
- Passing structures call by reference is more efficient than passing structures call by value as the latter requires an entire structure to be copied
- Example
  - Imagine that you have an array `people[100]` with each element of the type `person_info_t`
  - The following function scans the array and returns the information about the first person with the same height as input, or returns a null value if it cannot find any such person (`p` is the array containing the information and `np` is the number of elements in the array; `ht` is the height of the person)

```
person_info_t * find ( person_info_t * p, int np, int ht )
{
    int i;    /* Counter */
    for ( i = 0; i < np; i++ )
        if ( p[i].height == ht )
            return ( p+i );
    return ( NULL );
}
```

  - `NULL` is implicitly converted to type pointer to `person_info_t`; the explicit conversion would have the last statement written as

```
return ( ( person_info_t * ) NULL );
```

- Caution
  1. *Never assume that structures, like arrays, are automatically passed call by reference, and try to modify the caller's structure values in the called function*
  2. *Never attempt to assign a structure of one type to a structure of different type*

typedef

- Allows a programmer to define own variable types
- Can also be used to create synonyms (or aliases) for previously defined data types
  - There is no Boolean data type in C
  - Use `typedef` to declare a new data type

```
typedef int boolean;
```

  - Could have been achieved by

```
#define boolean int
```

  - `typedef` allows for the definition of more complex objects as well

```
typedef int group[10];
```

  Now, there exists a new type called `group` denoting an array of ten integers

```
/******************************************************************************/
/* Creating and initializing a variable array with enumerated type            */
/******************************************************************************/
int main()
{
    typedef int group[10];
    group totals;
    int i;
    for ( i = 0; i < 10; totals[i] = i++ );
}
```

  - Simple rule: The new type name is always the identifier on the right

- Mostly used with `struct` so a structure tag is not required
- Also good for portability of the code
  - A program requiring 4-byte integers may use type `int` on one system and type `long` on another system
  - Problem can be solved by using preprocessor commands to declare a type `integer` that will behave appropriately on either system

Nested structures

- Structures can be used to build more complex structures
- Let us think of a structure `employee_t` that holds information about employees in a company

```
typedef struct
{
    int         month;
    int         day;
    int         year;
} date_t;

typedef struct
{
    char        first_name[20];
    char        middle_init[2];
    char        last_name[20];
} name_t;

typedef struct
{
    name_t      name;               // Name of employee
    date_t      dob;                // Date of birth
    date_t      joining_date;       // Date when started with company
} employee_t;

typedef struct
{
    employee_t  employee;
    char        dept[20];
} manager_t;
```

Unions

- Derived data structure just like a `struct`, to hold one of more fields or attributes
- With structures, all members are physically present in the memory at the same time
- For some applications, this is not necessary because at each moment, only one of them contains useful information
- Consider the development of a package where the package has to keep space for different types of vectors but work with only one type at any time

```
struct vector
{
    int                 i;
    double              j;
    struct complex
    {
        double rl,
               im;
    }                   k;
};
```

- In general, this vector wastes space as only one of the members is required at any time
- Better to have variants rather than members such that the variants occupy the same memory space
- Accomplished by `unions`
- Only one member, and thus one data type, can be referenced at a time
  - The compiler will allocate space to hold the largest element in the union
- Above structure can be declared as

```
union vector
{
    int                 i;
    double              j;
    struct complex
    {
        double rl,
               im;
    }                   k;
};

union vector u, *p;
```

- As with structures, the following are valid:

```
u.i   u.j   u.(k.rl)
p->i  p->j  p->(k.rl)
```

- The sequence of statements

```
u.i = 123;
u.j = 3.14;
```

  results in the value in `u` to be `3.14` because `u.i` occupies the same space as `u.j`, or at least a part of it
- A union occupies as much memory space as its largest variant
- We can also declare functions that will return a union

```
union vector * read_vector ();
```

- Operations on unions
  - Assigning a union to another union of the same type
  - Taking the address (`&`) of a union
  - Accessing union members using the structure member operator and the structure pointer operator
  - In a declaration, a union may be initialized only with a value of the same type as the first union member

```
union vector vec = { 10 };
```

  - The following declaration will be invalid

```
union vector vec = { 3.14 };
```

- *Cautions*
  - *Referencing with the wrong type, data stored in a union with a different type, is a logic error*
  - *If data is stored in a union as one type and referenced as another type, the results are implementation dependent*
  - *Comparing unions is a syntax error because of the different alignment requirements on various systems*
  - *Initializing a union in a declaration with a value whose type is different from the type of the union's first member is an error*
  - *The amount of storage required to store a union is implementation dependent*

Bit Operations

- Bit or flag
  - Smallest unit of information
  - Can take on the value 1 (true) or 0 (false)
  - Used to manipulate the bits of integral operands, `char`, `short`, `int`, and `long`, both `signed` and `unsigned`
  - Used to control the machine at the lowest level, specially in pixel-level graphics
- Byte

  - Collection of 8 bits
  - Can be represented as two hexadecimal numbers by using `0xHH` where `H` is a hexadecimal number

- Bit operators
  - Allow a programmer to manipulate individual bits in integer or character data types

Operator	Semantics
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~	Complement
<<	Shift left
>>	Shift right

  - The `and` operator
    * Compares corresponding bits in the two operands
    * Consider the following program

```
main()
{
    char c1 = 0x45,
         c2 = 0x71;
    printf ( "Result of %x & %x = %x\n", c1, c2, c1 & c2 );
}
```

    * The operators `&` and `&&` are different
    * Using bitwise operator to check if a number is even

```
#define even(x) (((x) & 1) == 0)
```

  - The bitwise inclusive `or` operator
    * Compare two operands and set resultant bit to 1 if either of the corresponding bits is a 1
  - The bitwise exclusive `or` operator
    * Compare two operands and set resultant bit to 1 if either of the corresponding bits is a 1 but not both
  - The `not` operator
    * Also called one's complement, invert, or bit flip
    * Unary operator
    * Changes the corresponding bits to 0 if they are 1, or 1 if they are 0
  - The left and right shift operators
    * Used to move the data a specified number of bits
    * Bits shifted out of the left side disappear
    * New bits coming in from the right side are zeros
    * Example

```
/********************************************************************
/* Printing an unsigned integer in bits

#include    <stdio.h>
```

```
main()
{
    unsigned int x;                          /* Number to be printed
    void display_bits ( unsigned int );

    printf ( "Enter an unsigned integer: " );
    scanf ( "%u", &x );
    display_bits ( x );
}

void display_bits ( unsigned int value )
{
    unsigned int i,
                 display_mask = 1 << ( 8 * sizeof ( unsigned int ) -

    /* The display_mask contains 1 shifted by the number of bits in i

    printf ( "%7u = ", value );

    for ( i = 1; i <= ( 8 * sizeof ( unsigned int ) ); i++ )
    {
        putchar ( ( value & display_mask ) ? '1' : '0' );
        value <<= 1;

        if ( !( i % 8 ) )
            putchar ( ' ' );
    }

    putchar ( '\n' );
}
/*********************************************************************
```

- Declaration of bit fields or packed structures
  - Consider a structure with the following information:

`name`	$\leq$ 30 characters
`male`	1 or 0
`married`	1 or 0
`elderly`	1 or 0

  - The structure can be declared as

```
struct person
{
    char name[31];       /* 30 characters + end of string */
    unsigned male : 1,
             married : 1,
```

```
                elderly :1;
} people[1000];
```

- `male`, `married`, and `elderly` are bit-fields
  - The 1 following the colon indicates that each of these fields contains only one bit
  - These structure members can only have the values 0 or 1
  - Instead of 1, a greater number of bits may be chosen limited to the number of bits in a single machine word
- An assignment to bit fields can be made as

```
people[i].married = 0;
```

- Bit fields occupy little space in the memory
  - In the above example, `male`, `married`, and `elderly` may be bits of a single machine word
  - Bit fields have no address of their own, and so, we cannot use pointers to them
  - `&(person[i].married)` is not a valid expression
- Another example of a bit field or packed structure

```
struct item
{
    unsigned int list:1;        /* item is in the list      */
    unsigned int seen:1;        /* item has been seen       */
    unsigned int number:14;     /* item number              */
                                /* at most 16383 items      */
};
```

- Code to extract data from bit fields is relatively large and slow
- Better for human consumption than bit operators which are complex and error prone

Enumerated types

- Designed for variables that contain only a limited set of values
- Set of integer constants represented by identifiers or tags, known as *enumeration constants*
- Values in an enumeration start with 0, unless specified otherwise, and are incremented by 1
- Creating a new type `months`

```
enum months { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

- To number the months from 1 to 12, the enumeration is specified as

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DE
```

- The name of the `enum` type (`months` above) is optional and can be omitted
- Identifiers in an enumeration must be unique, and are generally written as upper case letters; they can be any valid C identifiers

- Value of each enumeration constant of an enumeration can be set explicitly in the definition by assigning a value to the identifier
- Multiple members of an enumeration can have the same integer value
- Using enumeration

```
/*************************************************************************
/* Using an enumeration type

#include <stdio.h>

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DE
typedef enum months month_t;

main()
{
    month_t month;
    char * month_name[] = { "", "January", "February", "March", "April", \
                            "May", "June", "July", "August", "September", \
                            "October", "November", "December" };

    for ( month = JAN; month <= DEC; month++ )
        printf ( "%2d%11s\n", month, month_name[month] );

    exit ( 0 );
}
/*************************************************************************
```

- *Cautions*
  - *Assigning a value to an enumeration constant after it has been defined is a syntax error*