Data types, `structs` and `unions` (Part II)

1 Character constants

- These are integer values that are written as a character in single quotes
- e.g. `'0'` = 48 in the ASCII character set
- These can also include escape characters:

`'\n'`	newline character
`'\a'`	alert (bell) character
`'\t'`	horizontal tab
`'\0'`	`NULL` character

- Example:

```
#define BELL '\a'
```

- On UNIX, you can run the `man ascii` command for more information. (Press `q` to exit.)

2 String constants

- These are zero or more characters in double quotes
- Technically this is an array of chars *and* it has a `NULL` character at the end of the string `'\0'`

```
char a[]="Hello";
char a[]={'H','e','l','l','o','\0'};
```

- This means that: `'x'` is not the same as `"x"` (i.e. `{'x','\0'}`)

```
#include<string.h>
char a[]="x";
char b='x';
strlen(a) = 1 // returns number of characters
sizeof(b) = 1 // returns number of bytes
sizeof(a) = 2
```

3 Getting user input

- To output text, we use `printf()`
  - e.g. `printf("%d",x);`
- To take user input from the user, we can use `scanf()`
  - e.g. `scanf("%d",&x);`
- Note the `&` (*address-of*) operator in front of `x`. Why do we need this?
- We can also take in options from the command-line e.g.

```
#include<stdio.h>
int main(int argc, char *argv[]){
  for(int i=0; i<argc; i++) {
    printf("Argument %d is %s\n",i,argv[i]);
  }
  return 0;
}
```

- `argc` is the number of command-line arguments and each `argv[i]` is a pointer to a string
- `argv[0]` is the name of the program
- Can convert strings to integers using `atoi()` from `stdlib.h` e.g.

```
char x[]="10";
int y=atoi(x);
```

4 Enumerations

- In many programs, we'll need variables that have only a small set of meaningful values
- A variable that stores the suit of a playing card should have only four potential values: "clubs", "diamonds", "hearts", and "spades"
- A "suit" variable can be declared as an integer, with a set of codes that represent the possible values of the variable:

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

- Problems with this technique:
  - We can't tell that `s` has only four possible values
  - The significance of `2` isn't apparent
- Using macros to define a suit "type" and names for the various suits is a step in the right direction:

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

- An updated version of the previous example:

```
SUIT s;
...
s = HEARTS;
```

- Problems with this technique:
  - There's no indication to someone reading the program that the macros represent values of the same "type"
  - If the number of possible values is more than a few, defining a separate macro for each will be tedious
  - The names `CLUBS`, `DIAMONDS`, `HEARTS` and `SPADES` will be removed by the preprocessor, so they won't be available during debugging
- C provides a special kind of type designed specifically for variables that have a small number of possible values
- An enumerated type is a type whose values are listed ("enumerated") by the programmer
- Each value must have a name (an enumeration constant)
- Enumerations are declared like this:

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

- The names of the constants must be different from other identifiers declared in the enclosing scope
- Enumeration constants are similar to `#define` constants directive, but not equivalent
- If an enumeration is declared inside a function, its constants won't be visible outside the function
- Behind the scenes, C treats enumeration variables and constants as integers
- By default, the compiler assigns the integers `0`, `1`, `2`, ... to the constants in a particular enumeration
- In the suit enumeration, `CLUBS`, `DIAMONDS`, `HEARTS` and `SPADES` represent `0`, `1`, `2` and `3`, respectively

5 Enumerations as Integers

- The programmer can choose different values for enumeration constants:

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

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

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

- It's even legal for two or more enumeration constants to have the same value
- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant
- The first enumeration constant has the value `0` by default
- 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`

6 Structures

- Collections of one or more variables forming a new data structure, the closest thing C has to an O-O class
- The elements of a structure (its *members*) aren't required to have the same type
- The members of a structure have names; to select a particular member, we specify its name
- In some languages, structures are called records, and members are known as fields

7 Structure: example

- For example a 2D point has `x` and `y` components but it is useful to create a single data structure to group them:
- Declares template for a point

```
struct point {
  int x;
  int y;
};
```

- With members `x` and `y`

8 Structures

```
struct point {
  int x;
  int y;
};
```

- Create an instance of the point data structure:

```
struct point a_point;
```

- Initialise a `struct`:

```
struct point a_point = {5, 6};
```

- Access to variable members of the structure:

```
a_point.x = 4;
a_point.y = 3;
```

9 Structure and scope

```
struct point {
  int x;
  int y;
};
```

- Each structure represents a new scope
- Any names declared in that scope won't conflict with other names in a program
- In C terminology, each structure has a separate name space for its members

10 Operations on structures

- The `.` used to access a structure member is actually a C operator
- It takes precedence over nearly all other operators
- Example:

  ```
  z = 20*a_point.x;
  ```

- The `.` operator takes precedence over the * operator

11 Assignment of structures

- The other major structure operation is assignment:

  ```
  point2 = point1;
  ```

- The effect of this statement is to copy `point1.x` into `point2.x`, `point1.y` into `point2.y` and so on
- The structures must have compatible types

12 Nested structures

- Declare a template for a rect(angle)

  ```
  struct rect{
    struct point pt1;
    struct point pt2;
  };
  ```

- Create an instance of the point data structure:

  ```
  struct rect a_window;
  ```

- Access to variable members of the structure:

  ```
  a_window.pt1.x = 4;
  ```

- What is the `sizeof(a_window)`?

13 Unions

- A `union`, like a structure, consists of one or more members, possibly of different types
- The compiler allocates only enough space for the largest of the members, which overlay each other within this space
- Assigning a new value to one member alters the values of the other members as well

13.1 Memory use

- The structure `s` and the union `u` differ in just one way
- The members of `s` are stored at different addresses in memory
- The members of `u` are stored at the same address

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

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

13.2 Accessing members

- Members of a union are accessed in the same way as members of a structure:

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

- Changing one member of a union alters any value previously stored in any of the other members
- Storing a value in `u.d` causes any value previously stored in `u.i` to be lost
- Changing `u.i` corrupts `u.d`

13.3 Properties

- The properties of unions are almost identical to the properties of structures
- Like structures, unions can be copied using the = operator, passed to functions and returned by functions

13.4 Initialisation

- By default, only the first member of a union can be given an initial value
- How to initialize the i member of u to 0:

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

13.5 Designated initialisers

- Designated initializers can also be used with unions
- A designated initializer allows us to specify which member of a union should be initialized:

```
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

13.6 For space saving

- Unions can be used to save space in structures
- Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog
- Each item has a stock number and a price, as well as other information that depends on the type of the item:
  Books: Title, author, number of pages
  Mugs: Design
  Shirts: Design, colors available, sizes available
- A first attempt at designing the `catalog_item` using `struct`:

```
struct s_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;
};
```

```
struct u_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;
};
```

13.7 Accessing nested structure

- This nesting of unions does make accessing the struct fields a little more complex:

```
struct s_catalog_item  c;

c.title

struct u_catalog_item c;

c.item.book.title
```

- Enumerations can be used to mark which member of a union was the last to be assigned

- In the number structure, we can make a kind member an enumeration instead of an `int`:

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

14 Using Enumerations to Declare "Tag Fields"

```
struct number a_number ={INT_KIND, {10}};

if (a_number.kind == INT_KIND)
  printf("a_number is %d value %d \n",
     a_number.kind, a_number.u.i );

a_number.kind = DOUBLE_KIND;

a_number.u.d = 150.03;
if (a_number.kind == DOUBLE_KIND)
  printf("a_number is %d value %6.3f \n",
     a_number.kind, a_number.u.d );
```

15 Creating new types

- `typedef` can be used to assign names to types

```
typedef unsigned char byte;
byte b1 = 12;
```

- You can use this with `structs` and `unions` too

```
typedef struct coords {
  int x;
  int y;
} point;
point p1={5,4};

typedef union id_thing {
  int i;
  double d;
} number;
number n = {.d =10.0};
```