Exercise 16 – More on Data Types

Objective

The objectives are to practice some basic bit manipulation (including an optional question), to create and manipulate a user-defined data union types, use typedefs and finally use function pointers with a jump table.

Reference Material

This is based entirely on the *Further Data Types* chapter. The practical session is located in the following directory:

Windows Directory:	**c:\qacprg\moredata**
Windows Solution directory:	**c:\qacprg\moredata\solution**
Linux Directory:	**/home/user1/qacprg/MOREDATA**
Linux Solution directory:	**/home/user1/qacprg/MOREDATA/Solution**

Overview

The first question is on bit manipulation and is a 'pencil and paper' exercise. The second and third questions are based on unions, the fourth is based on a `typedef` and the fifth is based on function pointers. The optional question is a more thought-provoking bit-manipulation exercise to be attempted only by bit enthusiasts!.

Practical Outline

The first question is on bit manipulation and is a 'pencil and paper' exercise. The second question is based on a `typedef`. The optional question is a more thought-provoking bit-manipulation exercise to be attempted only by bit enthusiasts!

1. What is displayed by the following code?

```
#define BIT0 0x1  /* bit 0 set (i.e. 00000000 00000001) */
#define BIT1 0x2  /* bit 1 set (i.e. 00000000 00000010) */
#define BIT2 0x4  /* bit 2 set (i.e. 00000000 00000100) */
#define BIT3 0x8  /* bit 3 set (i.e. 00000000 00001000) */
#define BIT4 0x10 /* bit 4 set (i.e. 00000000 00010000) */

int main(void)
{
   unsigned int i = 8, j = 11;
   printf("test1: %u\n", i & j);
   printf("test2: %u\n", i | j);
   printf("test3: %u\n", i | 0);
   printf("test4: %u\n", i & 0);
   printf("test5: %u\n", i << 2);
   printf("test6: %u\n", BIT0 << 8);
   printf("test7: %u\n", (BIT3 | BIT4) >> 2);
   printf("bit0: %s\n", j & BIT0 ? "ON" : "OFF");
   printf("bit1: %s\n", j & BIT1 ? "ON" : "OFF");
   printf("bit2: %s\n", j & BIT2 ? "ON" : "OFF");
   printf("bit3: %s\n", j & BIT3 ? "ON" : "OFF");
   printf("bit4: %s\n", j & BIT4 ? "ON" : "OFF");
   return 0;
}
```

Open the Visual Studio Solution **bits.sln** and build/run the program, to check your answers.

2. This question uses a `union` very similar to an example found in the course notes. Open the Visual Studio Solution **anytype.sln**, and take a look at the code template in **anytype.c**. `struct Anytype` contains an `int` and a `union Values`, i.e.:

```
struct Anytype
{
     int type;
     union Values value;      /* refer to course notes */
};
```

The `type` member contains `1, 2` or `3` if the `union` contains a `char`, `int` or `double` respectively. Using the example in the course notes of the *Structures* chapter as an example, write a function called `print_anytype` that has the following prototype:

```
    void print_anytype(const struct Anytype *);
```

This displays the value member of the `struct Anytype` in the current form, i.e.:

```
int main(void)
{
    struct Anytype a1;
    struct Anytype a2;
    struct Anytype a3;

    a1.type = 1;                /* Prepare for a char */
    a1.value.c_val = 'Z';

    a2.type = 2;                /* Prepare for a int */
    ...

    print_anytype(&a1);         /* Displays a char  */
    ...
}
```

A solution for this question is available in the **solution\anytype.sln** Visual Studio Solution.

3. For Windows, open the solution **view.sln**. For Windows and Linux take a look at the code template provided in **view.c**. The `union` definition presupposes that your `int` is a 4-byte quantity. *If you are on a machine that has a 2-byte `int`, please change the `c_view` array to an array of 2 `char`s.*

```
union View
{
    int   i_view;     /*  int interpretation  */
    char c_view[4];       /*  4 chars interpretation */
};
```

```
union View memchunk[20];        /* Not initialised */
```

Complete the program so that memory used for storing `memchunk` is displayed in both integer (4-byte `int`) and character (four 1-byte `chars`) form.

4. Open the Visual Studio Solution **tdef.sln**. Take a look at the header file, **tdef.h**, which contains prototypes for functions `Pinitialise` and `Ppoint`:

```
void Pinitialise(POINT *);
void Pprint(const POINT *);
```

`Pinitialise` assigns a value to a variable of type `POINT` by taking information from the user using the keyboard.

`Pprint` displays the contents of a `POINT` variable.

Now take a look at the source file, **tdef.c**. Complete the program by writing the definitions of the two functions. A solution for this part of the question is available in the Visual Studio Solution **solution\tdef1.sln**.

Alternative:

Instead of `Pinitialise` and `Pprint`, write functions called `Pscanf` and `Pprintf` that have format string arguments, so calls to these functions are:

```
Pscanf("%P", &mypoint);
Pprintf("%P", &mypoint);      /* Note the & */
```

A solution for this part of the question is available in the Visual Studio Solution **solution\tdef2.sln**.

Optional:

5. Open the Visual Studio Solution **prt_bits.sln**. Write a function that prints out the bit pattern of its parameter, i.e. with the prototype:

```
void print_bits(unsigned int);
```

Assume 8 bits per byte and a 4-byte integer