SCHOOL OF ENGINEERING

Software and Embedded Systems Laboratory 2 (ELEE08022)
C language part

Pointer & Data Operation

1. Variables and Addresses

All data used in a computer program are stored in the computer's memory. Inside the computer, a storage location is referred to by its address. We also saw that while the *computer hardware* uses addresses to reference storage locations. Since what is stored in a particular memory location (the content) can vary, we refer to a named memory location as a *variable* and the name is known as a *variable name*. By giving a variable a *name*, the programmer does not need to know or decide the address where data is physically stored (although the computer always knows!), but simply uses the variable name and lets the computer/compiler decide the physical location. However, we must tell the compiler the *type* of data to be stored in a *variable*, so that the appropriate size of memory area is allocated.

Although we will usually prefer to refer to variables by *name*, there are occasions when we may need, or choose, to refer to a variable using its *address*. In C, there is an easy way to do this: putting the unary *operator* **&** in front of a variable name gives the *address* of the variable. We have already seen the use of the **&** operator in calls to **scanf()**. The program below illustrates the difference between the *content* of a variable and its *address*.

Program 1
```
#include <stdio.h>
int main(void)
{
   int num;
   num = 22;
   printf("Contents of num = %d Address of num = %p\n", num, &num);
   return 0;
}
```

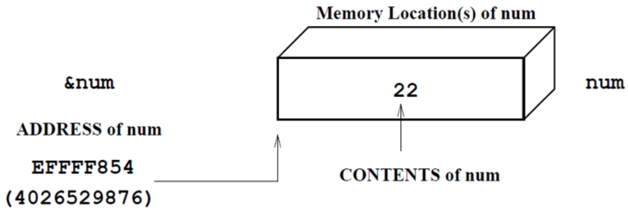
Running this program might give:

```
Contents of num = 22 Address of num = EFFFF854
```

The variable **num** was declared as **int**, so the usual decimal integer number format **%d** is used to print its content. The address of the same variable **(&num)** is printed using the format **%p**, where 'p' is for *pointer*, a word used to talk about addresses in the C language.

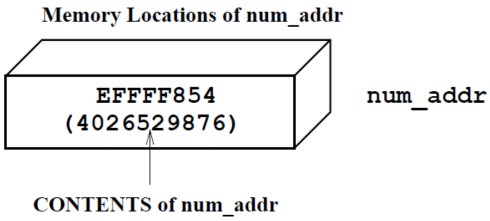
Since all information in a computer is in binary codewords, **%p** prints the pointer value (address) using *hexadecimal* notation, grouping the bits four at a time and representing each group as one of sixteen symbols, simply because binary notation would be very cumbersome for so many bits. The pointer could be printed in decimal number format, but there would be two immediate problems with doing that: (i) pointers (addresses) are not numbers; (ii) because of (i) it's not clear if we should decode the codeword as a signed or unsigned number value.

The particular address of the memory location used to store *num* will depend on the compiler and computer used to run the program above, and may even change from one run to the next. The range of possible memory addresses or *size* of a pointer, also depends on the type of computer used. For the 64 bits (8 bytes) computers, an enormous number of different memory locations to be addressed. Many computers use 32-bit addresses, and the vast number of microcontroller which controls things like toasters and car windows use 16 bits or fewer. In these notes, to keep things manageable, diagrams and examples will assume the use of 32-bit addresses (pointers). A diagrammatic representation of the computer memory storage occupied by a variable such as **num** is given below. The diagram also reminds us that the number of memory locations (usually called *bytes*) that are required to store a variable depends both on the *type* of the variable (**int**) in this case, and the particular computer being used.



2. Pointers

Since the address of a variable is simply a binary codeword, like a number or character value, we can store it in a suitably sized memory location (variable). We have declared variables to hold **char**acters, **int**egers, **float**ing point and **double** precision floating point values. We can equally well declare variables to hold *addresses*, such a variable being called a *pointer variable* (colloquially just "a pointer"). This is because the *address* which it holds tells us the location of i.e. *points* to the *other* variable. The diagram below illustrates what would happen, if we declared a pointer variable **num_addr** and used it to store the address of our original variable **num**.



Storing the address of the variable **num** in the pointer variable **num_addr** would, of course, be done using the **&** operator by assigning the address of **num** to the pointer **num_addr**:

```
num_addr = &num;
```

2.1. Using Addresses

We have now seen how we can get the address of a variable, and that we can store this address in a pointer variable, but why would we want to do this? The answer to **why** we want to do it will come later but let us fist look at **what** we can do with an address. As well as providing an *address operator* (**&**), C provides us with an *indirection operator* (*****). When an **&** immediately precedes a variable name, as in **&num**, this refers to the address where **num** is stored. When an ***** immediately precedes a pointer variable name, as in ***num_addr**, this refers to the *variable* whose address is stored in **num_addr**. Thus if the pointer variable **num_addr** contains the *address* of variable **num**, then referring to ***num_addr** in a program will have exactly the same effect as referring to **num** directly. Therefore ***num_addr** means *the variable whose address is stored in* **num_addr** or since **num_addr** is a *pointer* variable, we say *the variable pointed to by* **num_addr**. The ***** operator is called the *indirection* operator since the computer first fetches the contents of the pointer variable (eg. **num_addr**) to get the address of the desired variable and then uses this to fetch the value of the actual variable - an *indirect* operation.

2.2. Declaring Pointers

Pointer variables, like all other variables, must be *declared* before they are used. Just as we *must* specify the data *type* which an ordinary variable is to hold in a declaration statement, so we must specify the *type* of variable, which a pointer variable is to point to. Thus, if we require a pointer variable called **num_addr** to point to (hold the address of) an integer we would declare it as:

```
int *num_addr;
```

This should be read as, **num_addr** *is a variable, which can hold the address of an integer* or **num_addr** *is a pointer to an integer*. Note that this declaration is identical to the declaration of an integer *except* for the *****. The ***** is required since if **num_addr** is a *pointer* to an integer then ***num_addr** is an integer. We can declare pointers to **any** *type* of variable in a similar manner:

```
float *float_point;    /* float_point is a pointer to a float */
char *char_point;   /* char_point is a pointer to a character */
double *double_ptr;    /* double_ptr is a pointer to a double */
```

Thus, we say **float_point** is a pointer to a **float**, **char_point** is a pointer to a **char** and **double_ptr** is a pointer to a **double**.

The program below illustrates the declaration of a pointer to an integer (**num_addr**) and shows how the address of an integer is assigned to it (using the **&** operator). It also illustrates the use of the indirection operator ***** to access the variable pointed to by (**num_addr**).

Program 2

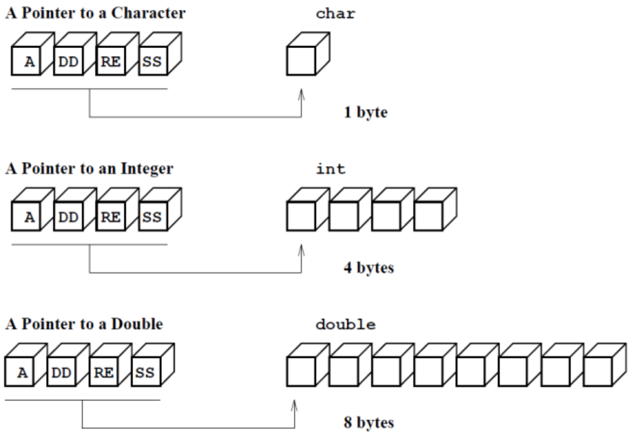
```
/* A program to illustrate the use of a pointer variable */
#include <stdio.h>
int main(void)
{
  int *num_addr;          /* declare a pointer to an integer */
  int num1, num2;                    /* declare two integers */

  num1 = 65;                             /* store 65 in num1 */
  num2 = 12345;                       /* store 12345 in num2 */

  printf("Address of num1: %p\n",   &num1);
  num_addr = &num1;     /* store address of num1 in num_addr */

  printf("Address stored in num_addr is %p\n", num_addr);
  printf("Value pointed to by num_addr is %d\n\n", *num_addr);

  num_addr = &num2;     /* store address of num2 in num_addr */

  printf("Address now stored in num_addr is %p\n", num_addr);
  printf("Value now pointed to by num_addr is %d\n\n", *num_addr);

  *num_addr = 54123;   /* store 54123 in *num_addr i.e. num2 */

  printf("Address now stored in num_addr is %p\n", num_addr);
  printf("Value of num2 is now %d\n", num2);

  return 0;
}
```

The output of the above program is:

```
Address of num1: 0028FF28
Address stored in num_addr is 0028FF28
Value pointed to by num_addr is 65
Address now stored in num_addr is 0028FF24
Value now pointed to by num_addr is 12345
Address now stored in num_addr is 0028FF24
Value of num2 is now 54123
```

Note that we **must** *initialise* a pointer, so that it contains a valid *address* **before** we use it to access a variable i.e. it must be made to point to a valid storage location!

Recall that variables of different *type* are usually of different size, or the data they hold may be encoded in different ways. Hence, a pointer declared to be pointer to one *type* of variable (e.g. an **`int`**) **must (almost) never** be used to point to a different *type* of variable (e.g. a **`float`**). Consequently, all pointers are of *type* "pointer to something", where "something" is one of the other types, which we have already encountered. Those "somethings" include the obvious **`int`**, **`char`**, **`float`**. As you can see from the figure below, the pointer is always pointing to the **first** memory storage location (byte).



2.3. `scanf()` re-visited

This technique of using pointers (or addresses) is the only means (apart from the generally very bad idea of using global variables, or the use of aggregates such as arrays and structures) by which we can effectively return *more than one value* from a C function. In fact, we have been making extensive use of this technique. Every time we have used the **`scanf()`** function we have passed it the *addresses* of the variable(s) in which we wanted it to store the values it read in.

```
int volts, current;

scanf("%d %d", &volts, &current);
```

This **`scanf()`** function call passes the *addresses* of two variables, **volts** and **current**, as the second and third arguments. The function uses these addresses to store values read from the keyboard. Thus the first integer value read in is stored in the variable whose *address* has been passed as the second argument to **`scanf()`** and the second value is stored at the address given in the third argument. This is the only way in which **`scanf()`** can return multiple values. So now we know *why* we *must* use an **`&`** immediately before variable names in a call to **`scanf()`**! If we forget the **`&`,** then **`scanf()`** will still assume the value we pass is an address (because that is what it has been written to expect, therefore, the value stored in the variable will be used *as an address* - with unpredictable and probably fatal results. Our C compiler is set up to warn us of this type of mistake - watch out for such *warnings*.

* When a function expects a pointer or address - remember to give it an address!

* When a function does not expect an address DON'T give it an address.

Ignoring these **guidelines** will certainly lead to problems!!

3. Using Pointers to Return Multiple Function Values

As we saw in Writing Your Own Functions session, a *called* function *always* receives a *copy* of the *value stored in the variable passed as an argument*. This method of calling a function and passing values to it is referred to as *call by value* since only the *value* of the argument is passed to the function. Inside the called function, we only have access to a *local variable* (the corresponding formal parameter) , which is *initialised* to the value of the variable given as an argument (in the calling function).

To illustrate how this works (again) we will consider a program intended to swap the values of two variables. A simple **`main()`** function to accomplish this *without using any function* is given below:

Program 3

```
/* Program to swap the values stored in two variables – Solution 1*/
#include <stdio.h>
int main(void)
{
 int a, b;                          /* two integers to hold values */
 int temp;                           /* temporary storage for swap */
 a = 2;
 b = 9;
 printf("Initially: Value in a = %d , Value in b = %d\n", a , b);
   /* Now do the swap by storing:- a in temp; b in a; temp in b */
 temp = a;
 a = b;
 b = temp;
 printf("Finally: Value in a = %d , Value in b = %d\n", a , b);
 return 0;
}
```

This operates in a very straightforward manner to produce:

```
 Initially: Value in a = 2, Value in b = 9
 Finally: Value in a = 9, Value in b = 2
```

as would be expected. If we were required to carry out a swap operation many times in our program, we might wish to write a function to carry out this task. Since a C function can only return *one* value it is not immediately obvious how we can carry out the desired swap using a function. One apparently attractive (but **COMPLETELY WRONG**) approach would be to write our function as follows.

Program 4

```
/* Program to swap the values stored in two variables - Solution 2*/
/* This version uses a function called swap() to do the swapping */
/* ****** THIS VERSION DOES NOT and CANNOT WORK CORRECTLY ****** */

#include <stdio.h>
int main(void)
{
  int a, b;                         /* two integers to hold values */
  void swap(int, int);              /* function prototype for swap */
  a = 2;
  b = 9;
  printf("Initially: Value in a = %d , Value in b = %d\n", a , b);
```

```
  swap(a, b);                              /* call the function swap */
  printf("Finally: Value in a = %d , Value in b = %d\n", a , b);
  return 0;
}
   /* Function to swap its arguments - This VERSION DOESN'T WORK */
void swap(int a, int b)            /* void since returns nothing */
{
  int temp;                        /* temporary storage for swap */
    /* Now do the swap by storing:- a in temp; b in a; temp in b */
  temp = a;
  a = b;
  b = temp;
  printf("In swap: Value in a = %d , Value in b = %d\n", a , b);
}
```

The results below simply confirm the accuracy of the comments in the program itself – i.e. it does not do what we wanted it to do:

```
Initially: Value in a = 2, Value in b = 9
In swap: Value in a = 9, Value in b = 2
Finally: Value in a = 2, Value in b = 9
```

Of course, the program does exactly what we asked it to do! The printout from inside the function (**`swap()`**) shows that our algorithm for swapping is correct. However, the variables called **`a`** and **`b`** in **`swap()`** are *local* variables of **`swap()`** (the formal parameters), whose only relationship to the **`a`** and **`b`** of calling function (**`main()`**), is that when **`swap()`** was called they were initialised with the contents of these variables. Thus altering the values of the **`a`** and **`b`** in **`swap()`** cannot have any effect on the values stored in the completely separate variables **`a`** and **`b`** in **`main()`**.

If we actually want a called function to alter variable values in the *calling* function then instead of passing *parameters* by *value* (as is normally the case), we must pass them by *reference*. The only way we can do this in C is by passing to the *called* function the **address** of the variable, rather than its *value*. The *called* function is then able to access the original variable (in the *calling* function), because it knows the variable's **address** (i.e. where its storage space is). By passing an address, we are effectively passing the content of a *pointer* to the variable.

To achieve this, we declare our new function **`swap()`** to take two arguments which are *addresses* or *pointers* to integers, rather than two integers, so the function prototype becomes:

```
void swap(int *, int *);      /* function prototype for swap */
```

When this function is *called* (in **`main()`**), **instead of passing `a`** and **`b`** as arguments we pass their *addresses* i.e. **`&a`** and **`&b`**. Thus the function call in **`main()`** becomes:

```
swap(&a, &b);              /* call the function - note two &s */
```

The function header for **`swap()`** is also altered to expect two addresses of integer variables:

```
void swap(int *p_a, int *p_b)  /* void since returns nothing */
```

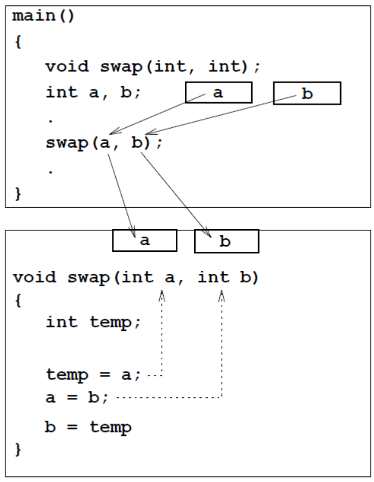
But how does this complicated procedure help us? As we have already discovered in this session, if the *pointer* **`p_a`** is initialised to the address of a variable **`a`**, then **`*p_a`** behaves as if it is the actual variable **`a`**. So when we write **`*p_a`** and **`*p_b`** in our new **`swap()`** function we can alter the variables **`a`** and **`b`**, even though they are outside the **swap()** function and declared as *local* (in the **main()** function). Applying this procedure, the following program correctly carries out the task we want.

```
Program 5
/* Program to swap the values stored in two variables - Solution 3*/
/* This version uses a function called swap() to do the swapping */
/* THIS VERSION PASSES ADDRESSES (pointers) so WORKS CORRECTLY */
#include <stdio.h>
int main(void)
{
  int a, b;                        /* two integers to hold values */
  void swap(int *, int *);         /* function prototype for swap */
  a = 2;
  b = 9;
  printf("Initially: Value in a = %d , Value in b = %d\n", a , b);
  swap(&a, &b);                    /* call the function - note &s */
  printf("Finally: Value in a = %d , Value in b = %d\n", a , b);
  return 0;
}

/* Swap function - swaps values at addresses supplied in arguments*/
/* N.B. Arguments are now addresses ie. pointers to variables */
void swap(int *p_a, int *p_b)      /* void since returns nothing */
{
  int temp;                        /* temporary storage for swap */
     /* Now do the swap by storing:- a in temp; b in a; temp in b */
  temp = *p_a;
  *p_a = *p_b;
  *p_b = temp;
  printf("In swap: Value in *p_a = %d, Value in *p_b = %d\n",
  *p_a, *p_b);
}
```

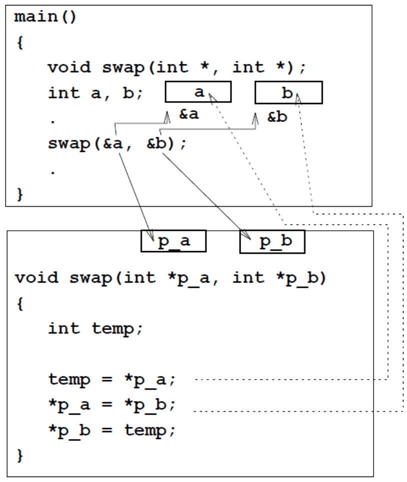
Running this program produces the correct results:

```
Initially: Value in a = 2, Value in b = 9
In swap: Value in *p_a = 9, Value in *p_b = 2
Finally: Value in a = 9, Value in b = 2
```

The differences between the incorrect and correct versions of the program are further illustrated in the figure below:



NON POINTER - WRONG Version.



POINTER - RIGHT Version.

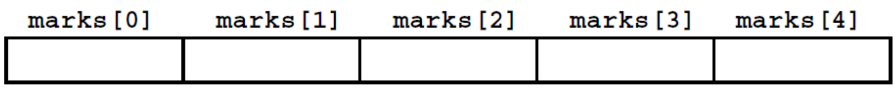
4. Arrays and Pointers

The address stored in a pointer gives *indirect* access to *another* variable. Pointers give an alternative method, other than **return**, by which to pass the result(s) of a function back to its calling environment. Hence, we may get back multiple values from a function by the use of pointers. Pointers are also used for other purposes in C programs and we will consider a few of these in this Session.

In C, there is a *very* close relationship between *arrays* and *pointers*. Consider a five element integer array **marks**, declared as:

```
int marks[5];
```

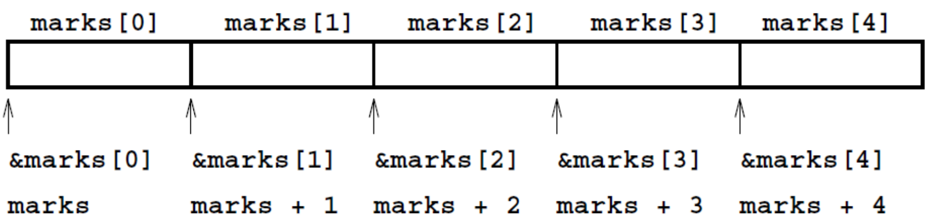
This is stored in the memory of the computer as shown here:



The individual elements are stored in memory in order of their index or subscript (0, 1, 2, …). In C, the *name* of an array (in this case **marks**) is actually a *pointer* to the array; specifically it holds the *address* of the first element (**marks[0]**) of the array. In fact, the computer interprets **marks** as being *equivalent* to **&marks[0]** - which is an expression (operator **&**) giving the address of the first element. The addresses of the other array elements are given by expressions:

&marks[1], **&marks[2]**, **&marks[3]** and **&marks[4]**

The C compiler, knowing the address of the first element, the *type* of the objects in the array, and arcane details of the hardware addressing scheme, can then create addresses (pointer values) to access each of the other individual array elements.



The figure shows an address expression for each of the elements of the array **marks**. (There may be gaps between adjacent elements of an array, which the compiler/hardware know about, but you don't/needn't/shouldn't). In a C program, the address of array element **marks[n]** is calculated as:

```
marks + n
```

as shown in the figure. This is how the compiler generates references to individual array elements, and shows that pointers are not numbers: **adding one** to a pointer gets you to the **next element**, no matter how big each element is, and even when there are gaps between elements (sometimes needed to keep the memory hardware happy).

By setting up a pointer variable (**m_ptr**) initialised with the starting address of an array (**&marks[0]**), the pointer can be used to access individual elements of the array as shown here:

Program 6

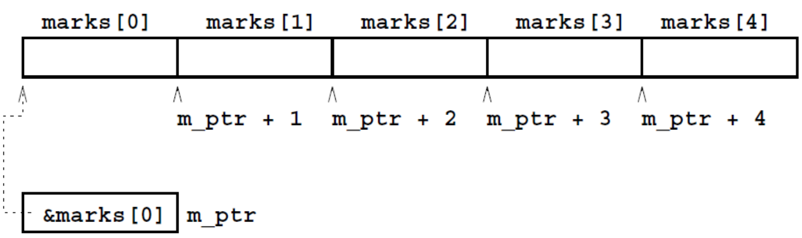
```
#include <stdio.h>
int main(void)
{
  int i, marks[] = { 81, 35, 72, 55, 19 };
  int *m_ptr;                 /* declare a pointer to an integer */
  m_ptr = &marks[0];/* assign starting address of array to pointer*/
  for (i = 0; i < 5; ++i)
  printf("marks[%d] = %d and *(m_ptr + %d) = %d\n",
                                        i, marks[i], i, *(m_ptr + i));
 }
```

Running this program shows the equivalence of subscripts (**marks[2]**) and pointers (***(m_ptr + 2)**)

```
marks[0] = 81 and *(m_ptr + 0) = 81
marks[1] = 35 and *(m_ptr + 1) = 35
marks[2] = 72 and *(m_ptr + 2) = 72
marks[3] = 55 and *(m_ptr + 3) = 55
marks[4] = 19 and *(m_ptr + 4) = 19
```

NOTE: that the parentheses in ***(m_ptr + 2)** are *essential* as the *indirection operator* ***** has higher precedence than *any* of the arithmetic operators (including the *multiply operator* *****). Writing ***mptr + 2** (without the parentheses) would cause the computer to fetch the contents of the location pointed to by **m_ptr** and add **2** to that. As **m_ptr** points to **grades[0]** this would then give the value **83** - not the **35** which we intended!

The figure below shows the relationship (in computer memory) between **m_ptr** and **marks**.



Since, as we have already mentioned, the *name* of an array is actually a *pointer* to the array, we did not actually need to create the pointer variable **m_ptr** to hold the starting address of the array. We could have used the actual array name **marks**. Thus we could have written the above program as:

Program 7
```
#include <stdio.h>
int main(void)
{
  int i, marks[] = { 81, 35, 72, 55, 19 };
  for (i = 0; i < 5; ++i)
  printf("marks[%d] = %d and *(marks + %d) = %d\n",
                                    i, marks[i], i, *(marks + i));
}
```

with equivalent results
```
 marks[0] = 81 and *(marks + 0) = 81
 marks[1] = 35 and *(marks + 1) = 35
 marks[2] = 72 and *(marks + 2) = 72
 marks[3] = 55 and *(marks + 3) = 55
 marks[4] = 19 and *(marks + 4) = 19
```

This seems simpler, but it is important to note that, although the array name **`marks`** is a pointer, **WE CANNOT CHANGE ITS VALUE**. Also, it is incorrect to try to find the address of an array by using the address operator on the array name *alone*, i.e. **`&marks`** DOES NOT give the address of the array. The expression **`&marks[0]`** DOES give the address of the array, i.e. the address of the first element of the array.

As the last two programs have shown, array subscripts and pointers are almost identical. This means that if we have a pointer variable **`m_ptr`** then **`*(m_ptr + i)`** can also be written as **`m_ptr[i]`**. This works because the compiler automatically converts array subscripts to use pointer notation i.e. a starting address (c.f. array name) + offset (c.f. subscript). Of course, if **`m_ptr`** points to a simple variable rather than to an array, writing **`*(m_ptr + i)`** or **`m_ptr[i]`** is correct C syntax but is of dubious practical value (unless **`i`** is **`0`**, or you are exploring regions of memory which perhaps you shouldn't...).

5. Passing Array Addresses to Functions

As we saw in previous session, when a *whole* array is passed to a function, all that is actually passed is the *starting address* of the array. Now that we understand why addresses are so useful and how pointers work, we can more fully appreciate the reasons why the C compiler handles arrays in this way. Since the name of an array is a pointer (constant) to the start of the array, then we can simply use straightforward pointer notation in the *called* function to access the elements of the (original) array. Adopting this approach we can rewrite the **`find_biggest()`** function in previous session using pointers only, as:

Program 8
```
#include <stdio.h>
#define N_MARKS 5
int main(void)
{
 int scores[N_MARKS] = {5, 15, 42, 9, 28};/* declare & initialise */

 int find_biggest(int *, int);              /* function prototype */

 printf("The largest value is %d\n", find_biggest(scores, N_MARKS));
}

int find_biggest(int *a_p, int size)    /* find the greatest value */
{
  int *e_p, biggest;
```

```
  e_p = a_p + size;                /* e_p points one past the end */

  for (biggest = *a_p; a_p < e_p; a_p++)
   if (biggest < *a_p)
     biggest = *a_p;
  return biggest;                       /* return greatest value */
}
```

Using pointer notation is not inherently better than using array subscripts when accessing arrays. Indeed, if you are a beginner at C programming, you may find the use of subscripts more comfortable. However, there are some tasks for which pointer notation is much more convenient and powerful so it is important to become familiar with its use.

6. Pointers to Structures

We can declare a **pointer to** a structure and use this to access the structure just as we have done with other types of C variables. Indeed pointers are very commonly used to access structures. For example, if the structure name tag **`student`** we have used above has been declared, then writing:

```
 struct student student1, *stu_ptr;
```

defines a **`student`** structure called **`student1`** and a **pointer** to a **`student`** structure called **`stu_ptr`**. We could make **`stu_ptr`** point to the actual structure **`student1`** by the following assignment:

```
 stu_ptr = &student1;
```

This uses the normal address operator **`&`** to get the *address* of the structure **`student1`**. As **`stu_ptr`** now contains the address of **`student1`**, so **`*stu_ptr`** refers to the actual structure **`student1`**. Thus we can either refer to the matriculation number stored in this structure as **`student1.matric_no`** or **`(*stu_ptr).matric_no`**. Note that the parentheses around **`*stu_ptr`** in this expression are **essential** because of the very high precedence of the structure member operator **`.`** (dot).

The use of pointers with structures is so common that a special **operator**, **`->`** (a hyphen **`-`** followed by a right angle bracket **>**) is provided to access structure members from a structure pointer. Thus, if **`stu_ptr`** is a pointer to the structure **`student1`** (as above) we can assign a value to the **`matric_no`** member by **either** of the following three statements:

```
    student1.matric_no = 9425604;
  (*stu_ptr).matric_no = 9425604;
    stu_ptr->matric_no = 9425604;
```

The notation for accessing structure members through a pointer to the structure, illustrated by the third of these statements, should be considered the *usual* means of handling the members of a structure. **Program 9** below uses pointer notation to access the structures. It demonstrates the convenience of using pointer notation to access individual structures, and their members, in an array of structures.

Program 9

```
#include <stdio.h>
#define NSTUD 5                      /* number of students in group */
#define SNLEN 22                           /* longest name expected */

struct student {       /* declare template for student structure */
 char name[SNLEN + 1];         /* student name held as a string */
```

```
 long matric_no;          /* matriculation number - must be long */
 int course_code;              /* course code held as an integer */
 int course_year;                   /* year of course e.g. 1 - 5 */
 char study_mode;                /* full time 'F', part time 'P' */
};
int main(void)
{
 struct student *stu_ptr;   /* a pointer to a student structure */
                          /* define & initialise structure array */
 struct student students[NSTUD] =
 {{"A Student", 1301023, 8413, 2, 'F'},
 {"A N O Student", 9301429, 8413, 2, 'F'},
 {"N X T Student", 314945, 8402, 2, 'F'},         /* NOT 0314945 */
 {"A P T Student", 1123467, 9300, 2, 'P'},
 {"T H E Lastbut-Notleast", 8721732, 8413, 2, 'F'} };

 printf("Name Matric No Course Year F/PT\n");
 for (stu_ptr = students; stu_ptr < students + NSTUD; ++stu_ptr) {
   printf("%-22s %07ld %4d %2d %c\n",
   stu_ptr->name,
   stu_ptr->matric_no,
   stu_ptr->course_code,
   stu_ptr->course_year,
   stu_ptr->study_mode);
   }
}
```

7. Structures and Functions

Individual structure **members** can be passed to a function like any ordinary variable. Most C compilers also allow a **complete structure** to be *passed to* or *returned from* a function, but beware! In the case of structures, the **whole** structure is *copied* (call by value), just like an ordinary variable and *unlike whole arrays*, where only a pointer value is passed (e.g. by giving the *array name*).

Program 10

```
#include <stdio.h>        /* copy and pass a structure to function */
 struct employee            /* declare a global structure template */
  {  int ind_num;
     double pay_rate;
     double hours;
  };

 int main (void)
 {
  struct employee temp = {6782, 8.93, 40.5};
  double net_pay;
  double calc_net (struct employee);        /* function prototype */

  net_pay = calc_net (temp);  /* pass copy of complete structure */
  printf("The net pay for employee %d is £%6.2f.",temp.ind_num,
                                                        net_pay);
  return 0;
 }
                           /*temp is of data type struct employee */
 double calc_net (struct employee temp)
 {
```

```
    return (temp.pay_rate * temp.hours);
  }
```

An alternative way passing a copy of a structure is to pass the address of the structure, which allows the called function to make changes directly to original structure.

Program 11

```
#include <stdio.h>           /* pass structure pointer to function */
 struct employee            /* declare a global structure template */
  {   int ind_num;
      double pay_rate;
      double hours;
  };
 int main (void)
 {
  struct employee temp={6782, 8.93, 40.5};
  double net_pay;
  double calc_net (struct employee *);      /* function prototype */

  net_pay = calc_net (&temp);        /* pass address of structure */
  printf("The net pay for employee %d is £%6.2f.",temp.ind_num,
                                                       net_pay);
  return 0;
 }
                  /* pointer pt is of data type struct employee */
 double calc_net (struct employee *pt)
  {
    return (pt->pay_rate * pt->hours);
  }
```