Software and Embedded System Lab 2 (ELEE08022)

Pointer & Data Operation in C language

Dr Jiabin Jia

Email: jiabin.jia@ed.ac.uk

What are we going to learn?

- What is pointer?
- Use pointer for variable and function
- Use pointer for array and function
- Use pointer for structure and function

Variables and Addresses

- Data in a computer program are stored in hardware *memory*
- Memory is identified by its *address*
- *Pointer in C language* is a type of variable which holds an *address*
- The unary *operator* **&** gives a variable's *address* from its *name*
  — we use **&** in front of the variable name – e.g. in **`scanf()`**

```
scanf("%f", &num); /take the address of num/
```

Pointers

- **A variable which holds the address of another variable is called a *pointer* variable**
- Pointer tells us the location of the other variable

Declaring Pointers

Pointer variables must be ***declared*** before they are used

- e.g. declare pointer variable to (point at/hold address of) an integer:

```
int *num_addr;
```

i.e. **num_addr** *is a variable which holds the address of an integer*

Declare pointers to any *type* of variable in a similar manner:

```
float  *f_ptr;       /* put _ptr on name to remind us! */
char   *c_p;        /* or _p is also a common reminder */
double *d;   /* still a pointer to double, but no hint */
```

Thus, we say:

f_ptr is a pointer to a **float**
c_p is a pointer to a **char**
d is a pointer to a **double**

```
#include <stdio.h> /*Program demonstrates the use of pointer variable*/
int main(void)
{
  int *num_addr;                      /* declare pointer to 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 num1 address in num_addr */
  printf("Address stored in num_addr: %p\n",   num_addr);
  printf("Value pointed to by num_addr: %d\n",*num_addr);

  printf("Address of num2: %p\n",   &num2);
  num_addr = &num2;                 /* store num2 address in num_addr */
  printf("Address stored in num_addr: %p\n",   num_addr);
  printf("Value pointed to by num_addr: %d\n",*num_addr);

  *num_addr = 54123;                /* store 54123 in *num_addr - num2*/

  printf("Address stored in num_addr: %p\n", num_addr);
  printf("Value of num2 is now: %d\n",       num2);
  return 0;
}
```

The output of above program is

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

Pointers and Multiple Value Returns from Functions

```
/* 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: a = %d , b = %d\n", a , b);
  temp = a;                            /* store a in temp */
  a = b;                              /* now store b in a */
  b = temp;                    /* finally store temp in b */
  printf("Finally:   a = %d , b = %d\n", a , b);
  return 0;
}
```

Run this program and print output

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

```c
#include <stdio.h>      /* Solution 2 - NOT really working */
int main(void)
{
  int a, b;                 /* two integers to hold values */
  void swap(int, int);  /* parameters are LOCAL variables! */

  a = 2;
  b = 9;
  printf("Initially: a = %d , b = %d\n", a , b);
  swap(a, b);           /* arguments are VALUES of a and b */
  printf("Finally:   a = %d , b = %d\n", a , b);
  return 0;
}
void swap(int a, int b)       /* THIS VERSION DOESN’T WORK */
{
  int temp;                  /* temporary storage for swap */
  temp = a;                             /* store a in temp */
  a = b;                               /* now store b in a */
  b = temp;                     /* finally store temp in b */
  printf("In swap:   a = %d , b = %d\n", a , b);
}
```

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

Passing Pointer Arguments to Function

By passing a variable's **address** to a function, rather than its *content*:

- function can access the ***actual variable***, not just a copy of its value, because function knows where to find the variable! (at its **address)**
- The word "pointer" in C language means "address"

So declare **`swap()`** to take arguments which are ***pointers*** - giving the *locations* of the variables - rather than the *values* in those locations.

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

swap(&a, &b);          /*call function passing pointers*/

void swap(int *p_a, int *p_b)/*function header*/
```

Now, in function **`swap()`**:

`*p_a` *refers to* the integer variable **`a`** in **`main()`** and

`*p_b` *refers to* the integer variable **`b`** in **`main()`**

So **`swap()`** *directly* uses **`main()`**'s variables **`a`** and **`b`** through pointers

```
#include <stdio.h>/*Solution 1- Working now by pointers*/
int main(void)
{
  int a, b;             /* two integers to hold values */
  void swap(int *, int *); /* function has pointer args*/
  a = 2;
  b = 9;
  printf("Initially: a = %d , b = %d\n", a , b);
  swap(&a, &b);            /* addresses passed to swap */
  printf("Finally:   a = %d , b = %d\n", a , b);
}
void swap(int *p_a, int *p_b)       /* function header */
{
  int temp;             /* temporary storage for swap  */
  temp = *p_a;           /* store value from a in temp */
  *p_a = *p_b;          /* now store value from b in a */
  *p_b = temp;              /* finally store temp in b */
  printf("In swap: *p_a = %d, *p_b = %d\n", *p_a , *p_b);
}
```

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

Arrays and Pointers

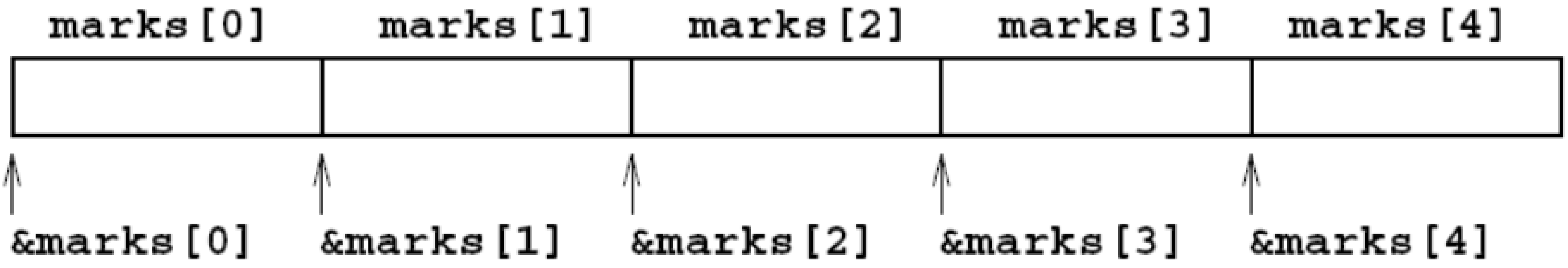
- Arrays and Pointers have a very close relationship
- Array elements stored in memory in *subscript (index) order*
- Change pointers to access array elements

Consider **`marks`**, a five element integer array:

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

The addresses of its elements are:

`&marks[0], &marks[1], &marks[2], &marks[3], &marks[4]`



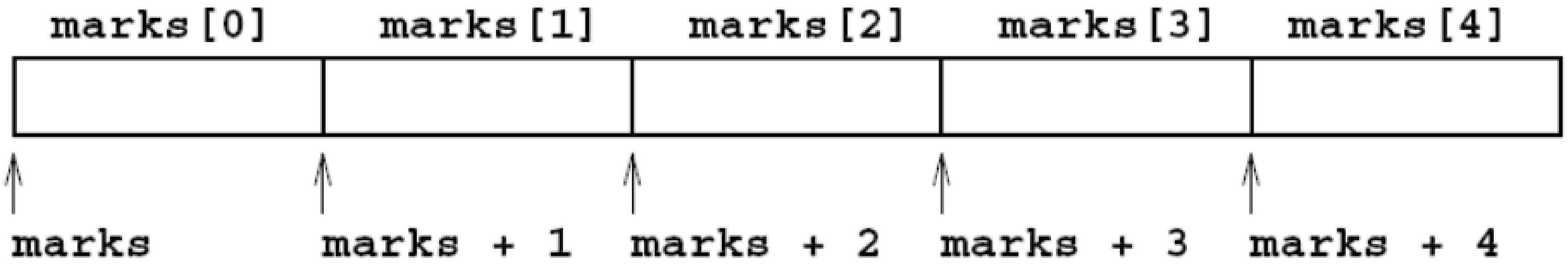
Name of an array in C (e.g. **marks**) is a *pointer* to the array

Using Pointers to Access Array Elements

- Consider integer array **`marks`** with 5 element

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

- Address of the first element is **`marks`** *(no & is needed)*
- Address of array element **`marks[n]`** is calculated as **`marks + n`**



- Declare a pointer variable of appropriate type (e.g. **`int *m_ptr`**)
- Initialise it with the starting address of the array

  (e.g. **`m_ptr = &marks[0] or m_ptr = marks`**)
- Use the pointer to access element **`n`** of the array: **`*(m_ptr + n)`**

```
#include <stdio.h> /* Use pointer to access the elements*/
int main(void)                            /*in an array */
 {
  int i, marks[5] = { 81, 35, 72, 55, 19 };
  int *m_ptr;                       /* pointer to an integer */

  m_ptr = &marks[0];                   /* initialise pointer */

  for (i = 0; i < 5; ++i) {
   printf("marks[%d] = %d and *(m_ptr + %d) = %d\n",
                 i,  marks[i],          i,  *(m_ptr + i));
    }
 }
```

```
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: *essential* parentheses in expression `*(m_ptr + i)`

indirection operator `*` has higher precedence than arithmetic operators

`*m_ptr + i` would add value of `i` to value of *first* element of array

Using Pointers to Access Array Elements

We could have written the previous program as:

```
#include <stdio.h>               /* Alternative way */
int main(void)
{
  int i, marks[5] = { 81, 35, 72, 55, 19 };
                          /* no pointer is declared */
  for (i = 0; i < 5; ++i)
  printf("marks[%d] = %d and *(marks + %d) = %d\n",
                 i,   marks[i],         i,   *(marks + i));
  return 0;
}
```

```
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
```

NOTE `*(marks + i)` is equivalent to `marks[i]`
`*(m_ptr + i)` is equivalent to `m_ptr[i]`

Passing an Array to a Function Using Pointer

When a *whole* array is passed to a function, what is *actually* passed is the *starting address* of the array (recall previous session).

The name of an array is a (constant) pointer to the start of the array

- So use pointer notation in *called* function to access single elements

```
#include <stdio.h>                                     /* Program 11_5 */
int main(void)  /* Actually Program 8_5 using pointers already! */
{
  int scores[5] = {5,15,42,9,28};              /* declare & initialise */
  int find_biggest(int *, int);                  /* function prototype */
  printf("Biggest value: %d\n", find_biggest(scores, 5));
  return 0;
}

int find_biggest(int *a_p, int size)                   /* find biggest */
{
  int *e_p, biggest;
  e_p = a_p + size - 1;          /* pointer e_p points to array end */

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

Pointers to Structures

- Pointer to structure similar to other pointers – memory address — passing/copying pointers around a program is efficient

Suppose that structure name tag student has been declared. Then:

```
struct student a_student, *stu_ptr;
```

- declares a_student a student structure
- declares stu_ptr a **pointer** to a student structure

```
stu_ptr = &a_student;
```

- *stu_ptr now refers to the *actual* structure a_student
- *matriculation number* member can be accessed as

```
(*stu_ptr).matric_no
```

Parentheses essential*:* precedence of member operator . (dot)

Pointers to Structures

Used so often they have a special C **operator ->**
(hyphen - followed by right angle bracket >):
The following statements are therefore all equivalent:

```
  a_student.matric_no = 1725604;
(*stu_ptr).matric_no  = 1725604;
  stu_ptr->matric_no  = 1725604;
```

```
#include <stdio.h>                               /* pointers */

struct student {                       /* structure template */
  char name[40+1];                  /* student name - string */
  long matric_no;                   /* matriculation - long  */
  int  course_code;                 /* course code - integer */
  int  course_year;                 /* course year eg. 1 - 5 */
  char study_mode;                  /* full/part time 'F'/'P'*/
};

#define NSTUD 5        /* NSTUD is no of students in group */
int main(void)
{
  struct student *stu_ptr;/* stu_ptr points to student struct */
  struct student students[NSTUD] =
  . . .
```

```
#define NSTUD 5   /* NSTUD is the number of students in group */
int main(void)
{
  struct student *stu_ptr;/* stu_ptr points to student struct */
  struct student students[NSTUD] =
  {
    { "A Student",     1601023, 8413, 2, ‘F’ },
    { "A N O Student", 1601429, 8413, 2, ‘F’ },
    { "N X T Student", 1614945, 8402, 2, ‘F’ },
    { "A P T Student", 1623467, 9300, 2, ‘P’ },
    { "T H E Last",    1621732, 8413, 2, ‘F’ }
  };
  printf("Name             Matric No. Course Year F/PT\n");
  for (stu_ptr=students; stu_ptr < students+NSTUD; ++stu_ptr) {
    printf("%-20s %07ld  %4d  %2d    %c\n",
             stu_ptr->name,
             stu_ptr->matric_no,
             stu_ptr->course_code,
             stu_ptr->course_year,
             stu_ptr->study_mode);
    }
  return 0;
}
```

```
Name                 MatricNo Course Year F/PT

A Student             1601023  8413   2    F
A N O Student         1601429  8413   2    F
N X T Student         1614945  8402   2    F
A P T Student         1623467  9300   2    P
T H E Last            1621732  8413   2    F
```

Passing Structures to Functions

- Individual structure members may be passed to a function as any scalar variable. For example, given the structure definition

```
struct { int id_num;
         double pay_rate;
         double hours;
       } temp;
```

- Pass a copy of the structure member temp.id_num to a function **`display ()`**

```
display(temp.id_num);
```

- Whole structure can also be passed to a function by including the name of the structure as an argument to the called function

```
calc_net(temp);
```

- But whole structure is copied!! Hard work for the machinery

Passing Structures to Functions

```
#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 copies of the values in temp */
  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);
  }
```

```
Program output
   The net pay for employee 6782 is £361.66.
```

Passing Structures to Functions using Pointers

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

```
double calc_net(struct employee *);/* function prototype */

calc_net (&temp);                         /* calling function */
```

- Function `calc_net ()` must declare the argument as a pointer

```
calc_net (struct employee *pt)      /* header of function */
```

Passing Structures to Functions using Pointers

```
#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 copies of the values in temp */
  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);
  }
```

```
 Program output
     The net pay for employee 6782 is £361.66.
```