C Programming Practice (32H)

@ Yi Chen
leo.chen.yi@live.co.uk

No1 - Introduction (2H)

No2 - Types, Operators And Expressions (2H)

No3 - Control Flow (6H)

- Statements and Blocks
- Conditional Statements
  - If-Else
  - Switch
- Loop
  - While
  - For

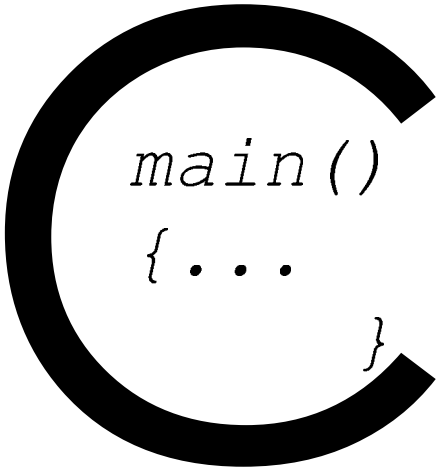
No4 - Functions And Program Structure (6H)

No5 - Pointers And Arrays (4H)

No6 - Case Studies And Practices (12H)

C Programming Practice
No[5-1]

Chen , Yi
leo.chen.yi@live.co.uk
30-Jul-2010



Pointer
– the memory address

BETA 1.0.0.1

Contents

Contents

What is a Pointer

One of C's **advantages** is that it allows the programmer **access to** the **addresses** of variables used in a program.

This gives the C programmer capabilities and programming power that is not available in other high-level languages.

A **pointer** is a variable or constant whose value **is the address** of another variable or constant.

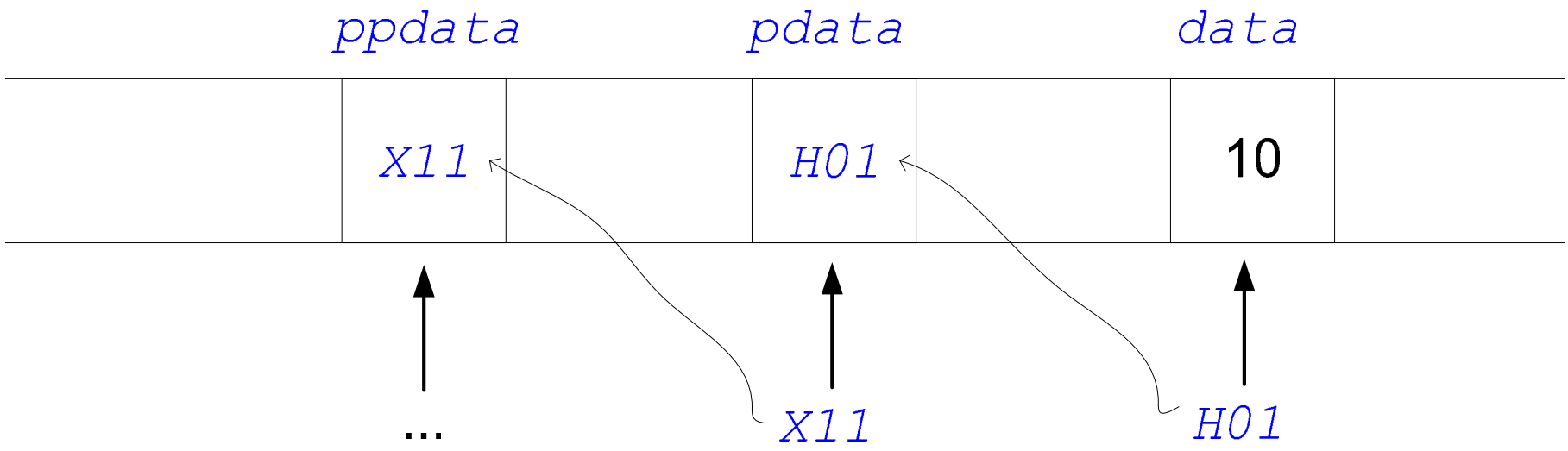
Modifying a **pointer value** simply **modifies** the **address** it points to.

To **modify** the **variable** that a pointer points to, it must first be **dereferenced** to **gain access** to the **actual variable**.

What is a Pointer

`*` - declare a pointer
`&` - address of a variable

```
int    data   = 10;
int   *pdata  = NULL;
int   **ppdata = NULL;

pdata  = &data;
ppdata = &pdata;
```



What is a Pointer

Every variable has **three** major items we need to pay attention to **à**

In C, a variable's data type is declared using a declaration statement.

1) its data type

float, int, char

2) the actual value stored in the variable

```
float velocity = 72.0;
int   week     = 11;
char  flag     = '1';
```

3) the address of the variable

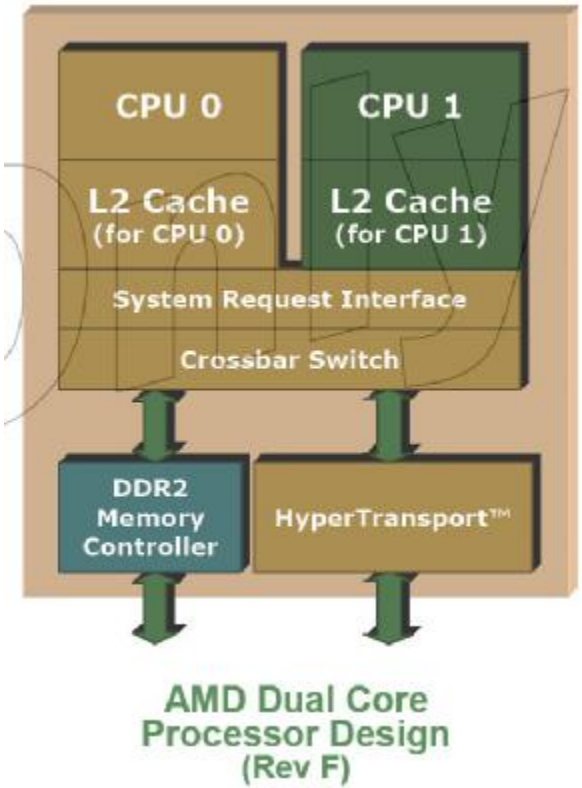
```
float time = 0.0;
float *time  = &time ;
```

What is a Pointer

```
 int print_pointer( )
{

int  week = 22;

printf( "week = %d,
          the address of week is %p", week, &week );

return 0;
}
```

Physical and virtual memory

- **Physical** memory: physical resources where data can be stored and accessed by your computer:
  - ***cache***[kash]
  - ***RAM*** - *Random-access memory*
  - ***hard disk***
  - ***removable storage***



- **Virtual** memory: abstraction by OS, addressable space accessible by your code

Physical and virtual memory

- **How much physical memory do I have?**

2 MB (cache) + 2 GB (RAM) + 100 GB (hard drive) + . . .

- **How much virtual memory do I have?**

< 4 GB (32-bit OS), typically 2 GB for Windows,

3-4 GB for linux

- **Virtual memory maps to different parts of physical memory**

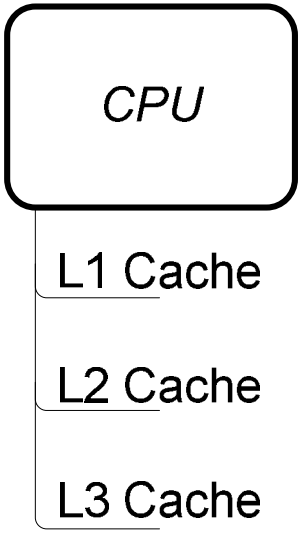
- **Usable parts of virtual memory: stack and heap**
- stack: where declared variables go
- heap: where dynamic memory goes

Physical and virtual memory

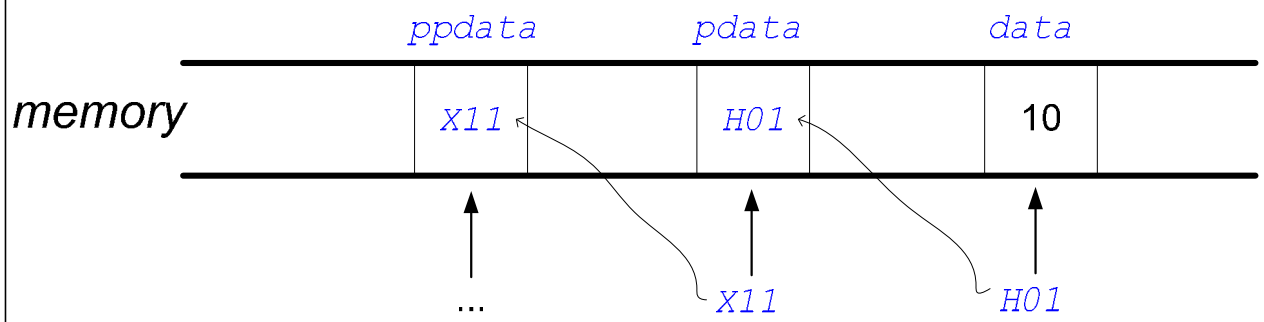
1) Depends on **OS** and **Hardware(CPU)**
2) concepts keep **updating**

A **buffer** is something that has yet to be "written" to disk.

A **cache** is something that has been "read" from the disk and stored for later use.



provided by hardware



provided by software, e.g. malloc()

Contents

Declaring a Pointer

The syntax of declaring a pointer is almost the **same** as the syntax of **declaring** the **variables** we have worked with in previous chapters.

The following statement declares an **integer** pointer variable:

```
int* flag = NULL;
```

The **asterisk (*)** you use to declare a variable as a pointer.

Either alternative syntax is equally correct because the compiler generally **ignores white spaces** between an operator and a variable name, constant name, or number.

Indeed, the following pointer declaration also works:

```
int* flag = NULL;
```

Declaring a Pointer

`*` - declare a pointer
`&` - address of a variable

```
int  data     = 0;
int  *pdata   = &data;
int  **ppdata = &pdata
```

The **syntax** of declaring a pointer is almost the **same** as **declaring** a **variable** which stores a value rather than an address.

However, the **meaning** of the **pointer's data type** is quite **different** than the meaning of the data type of a variable which stores a value rather than an address.

A **pointer itself** may be **a variable or a constant**, and like other variables or constants, it is also stored at a memory address. What distinguishes a pointer is that its value is the memory **address** of another variable or constant.

Declaring a Pointer

Pointer arrays

>> Pointer array – **array of pointers, each of its element is a pointer**

int *arr[20]; – an array of pointers to integer’s
char *arr[10]; – an array of pointers to character’s

>> Pointers in array-**each element-**can point to arrays themselves

char *strs[10]; – an array of char arrays (or strings)

Declaring a Pointer

String arrays

>> An array of strings, each stored as a **pointer** to an **array** of chars

>> Each string **may be** of **different length**

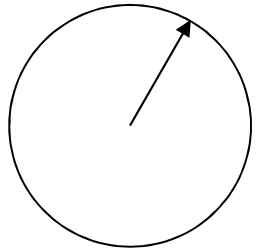
```
char  str1[ ]      =  "hello" ;    /* length  =  6 */
char  str2[ ]      =  "goodbye" ;  /* length  =  8 */
char  str3[ ]      =  "ciao" ;     /* length  =  5 */

char  strArray[ ]  =  { str1, str2, str3 } ; *
```

>> Note that **strArray** contains **only** pointers, not the characters themselves.

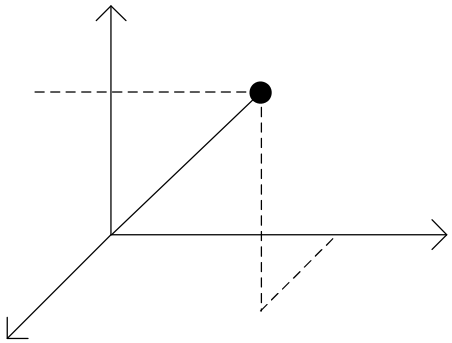
Declaring a Pointer

Multi-dimensional arrays

>> C also permits multidimensional arrays specified using **[ ]** brackets notation:



```
float  map[20][30];
/*is an 20x30 2-D array */
```



```
float  map[20][30][10];
/*is an 20x30x10 3-D array */
```

>> **Higher** dimensions possible:

```
char world3D[15][7][35][4];
```

– what are the dimensions of this?

Declaring a Pointer

- pointers:

```
int  xdata  = 0;
int *pxdata = &xdata;
```

- pointers to pointer:

To summarise à

```
int  xdata    = 0;
int *pxdata   = &xdata ;
Int **ppxdata = &pxdata ;
```

- Array of pointers:

```
char *names[]={"abba","u2"};
```

- Multidimensional arrays:

```
float  x[20][20] = 0.0;
```

Declaring a Pointer

- void pointers **can** be used to point to **any** data type

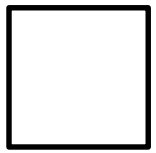
```
 int  xdata   = 0;
 void *pxdata = &xdata;
/* points to  int */

  float fdata  = 0.0;
  void *pfdata = &fdata;
/* points to float */
```

- void pointers cannot be **de-referenced**. The pointers should always be cast before dereferencing.

```
void *pdata;
printf ("%d", *p); /*  invalid  */

void*  pdata;
int   *pxdata = (int*)pdata;
Printf ("%d", * pxdata); /* valid */
```

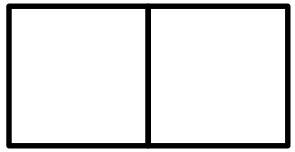
Declaring a Pointer

A pointer to a **character**



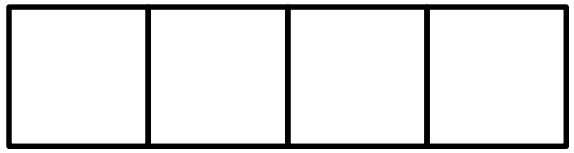
One byte is retrieved

A pointer to an **integer**



Two bytes are retrieved

A pointer to a **float**



Four bytes are retrieved

Contents

Assigning a Value to a Pointer

Assign a value to a pointer before you use it.

```
int   num      = 5;   int* inum  = &num;
float velocity = 0.0; float* fveo = &velocity;
```

When the variable is a **pointer**, that leftover **value** is interpreted as **another memory address.**

There are a number of memory address ranges that you are **not permitted** to access programmatically, such as those **reserved** for use by the **operating system**. If the leftover value is interpreted as one of those **prohibited addresses**, the result is **an error**.

Assigning a Value to a Pointer

*To **initialise** the pointers,*

e.g.
*int *pdata   = NULL;*

A pointer that is assigned **NULL** is called a ***null*** *pointer.*

In C, **function** itself is **not** a variable. **But** it is possible to declare pointer to functions.

Declaration examples:

- `int (*fp)( int )  /* notice the () */`
- `int (*fp)( void* , void* )`

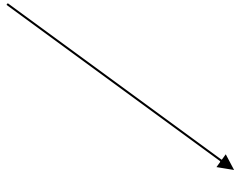
Contents

Indirection Operator and Dereferencing

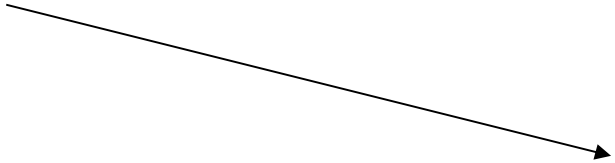
The primary use of a pointer is to **access** and, if appropriate, **change** the value of the variable that the pointer is pointing to.

e.g.

```
int  data  =  3;
```



```
/* pointer to data */
int *pdata =  &data ;
```



```
/* pointer to address of pdata  * /
int **ppdata =  &pdata ;
```

Contents

The Pointer as a Variable or a Constant

A pointer may be a **variable** or a **constant**. Let's examine both possibilities.

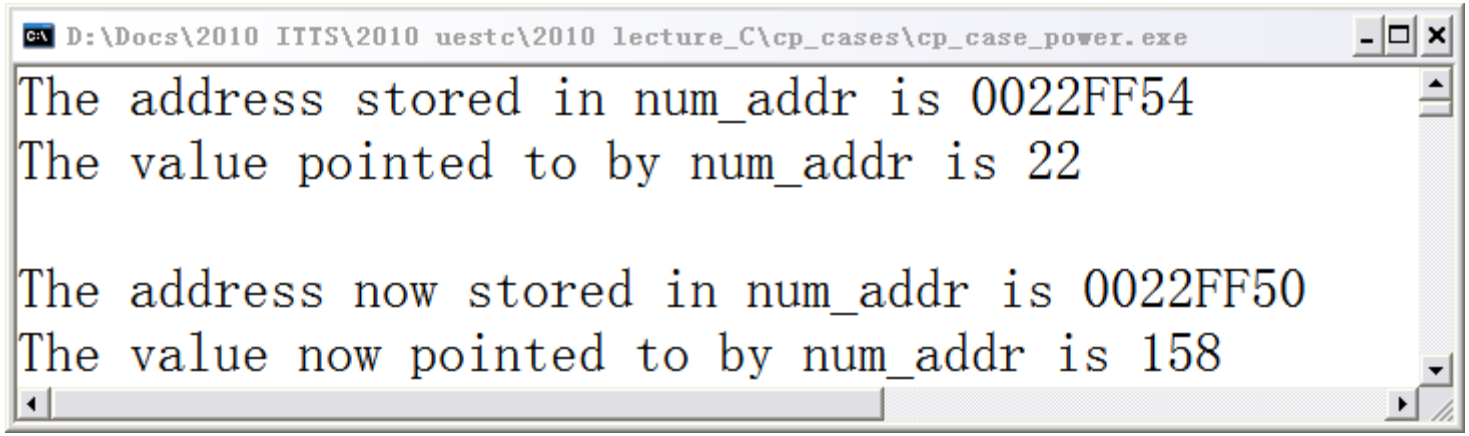
Case 1 : Pointer as a Variable

The preceding program had the pointer **pointing to** one integer variable. However, a pointer variable, being a variable, can point to **different** variables at **different** times in the program.

In the following program, the value of the pointer is changed to point to two different integer variables.

```
int pointer_as_variable(void)

{

/* declare two integer variables */
int    miles    = 22;
int    dist     = 158;

/* declare a pointer to an int*/
int   *num_addr = &miles;

printf("The address stored in num_addr is p\n",num_addr);
printf("The value pointed to by num_addr is %d\n\n",*num_addr);

num_addr = &dist;

/*now store the address of dist in num_addr*/
printf("The address now stored in num_addr is %p\n",num_addr);
printf("The value now pointed to by num_addr is
%d\n",*num_addr);

return 0;   }
```

Outputs of **pointer_as_variable**()



The Pointer as a Variable or a Constant

Case 2 : The Array Name as a Constant Pointer

While the pointer may be a variable, it also may be a constant.

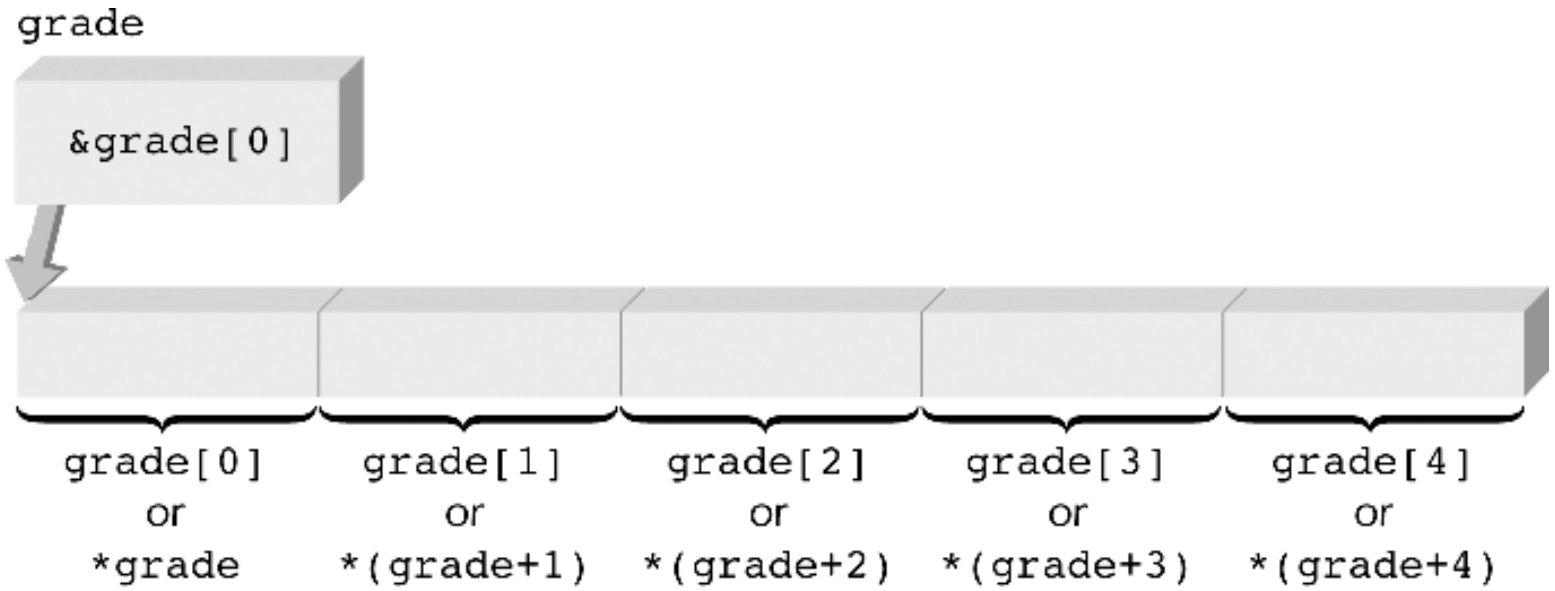
Indeed, the value of the **name** of an **array** is the base **address** of the array, which **also** is the **address** of the **first element** of an array.

Thus, in the following program, both ***testScore*** and ***&testScore[0]*** have the **same** value

```
int pointer_as_array_name(void)

{

int testScore[3] = {4, 7, 1};

printf("The address stored in num_addr is
%p\n",testScore );

printf("The address of the first element of the array "
      "using &testScore[0] is %p\n",&testScore[0]);

printf("The value of the first element of the array "
      "using *testScore is %d\n",*testScore );

printf("The value of the first element of the array "
      "using testScore[0] is %d\n",testScore[0]);

   return 0;   }
```

Outputs of **pointer_as_array_name()**

```
D:\Docs\2010 ITTS\2010 uestc\2010 lecture_C\cp_cases\cp_case_power.exe
The address stored in num_addr is 0022FF40
The address of the first element of the array using &testScore[0] is 0022FF40
The value of the first element of the array using *testScore is 4
The value of the first element of the array using testScore[0] is 4
```



Contents

Pointer Arithmetic

3 Cases

The value of a **pointer**, even though it is an address, **is** a **numeric value**. Therefore, you can perform **arithmetic operations** on a pointer just as you can for a numeric value.

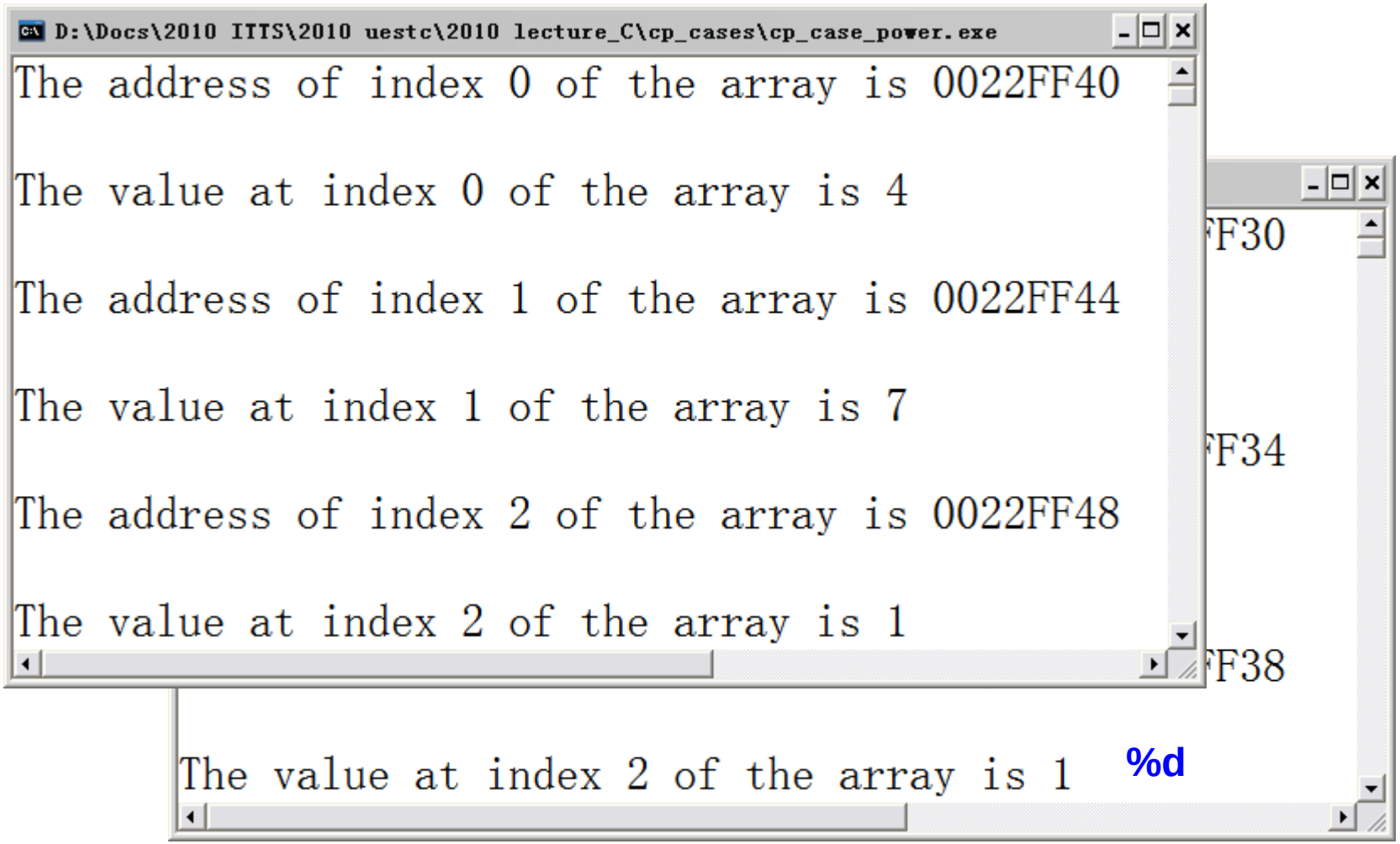
Case 1: Using a Variable Pointer to Point to an Array

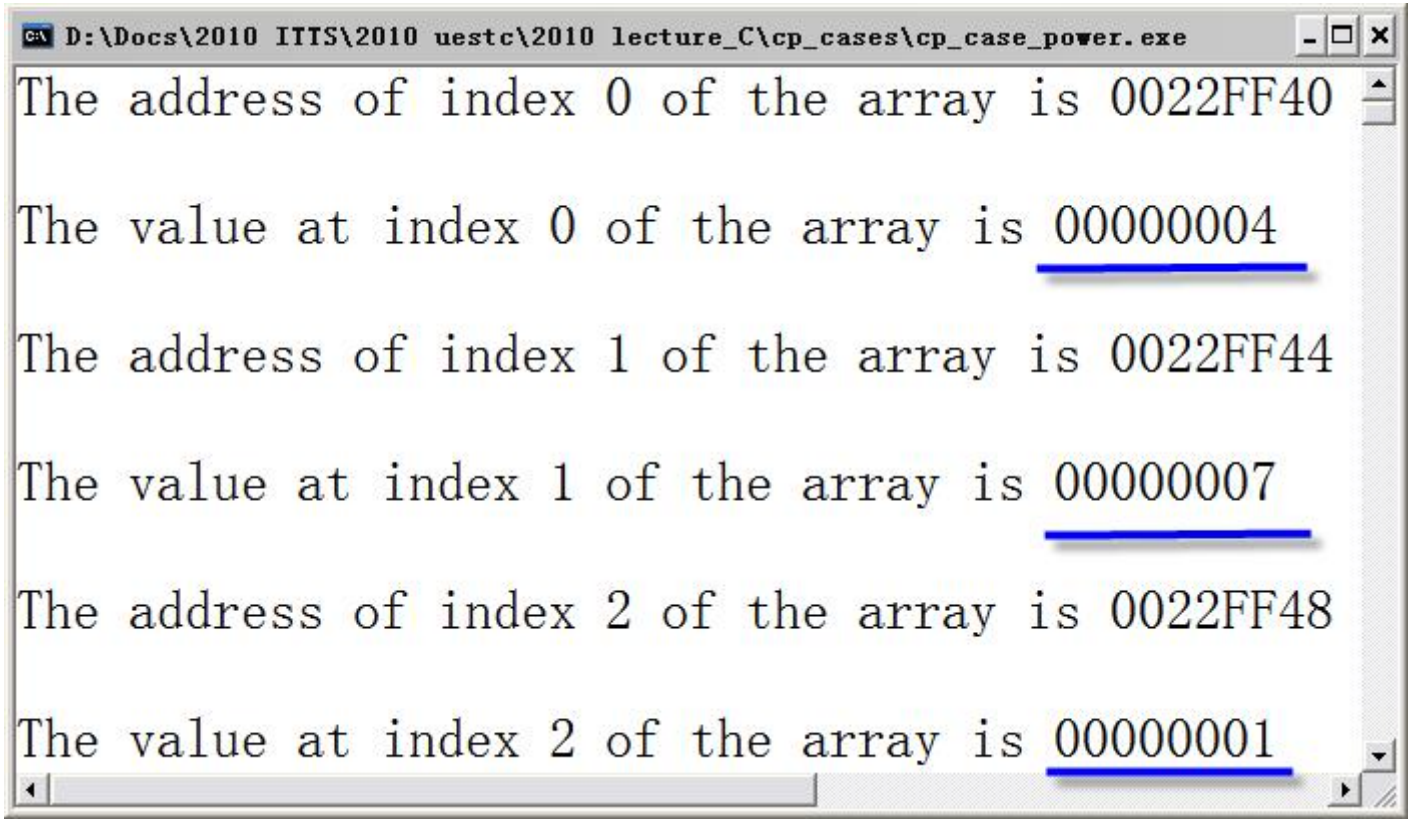
you **cannot change** the value of the **name** of **an array**, it being a **constant pointer**, you **first** should **declare** a **variable** pointer and then **assign** it to the address of an array.

So, we begin with an established point of reference, let’s start with the following program, which **outputs** the **address** and value at **each element** of an array using the name of the array:

```
int pointer_arithmetic_array(void)


{

   int testScore[3] = {4, 7, 1};
   int idx          = 0;

   for ( idx = 0; idx < 3; idx++ )
   {

    printf("The address of index %d of the array is
%p\n\n",idx,&testScore[idx]);

    printf("The value at index %d of the array is
%d\n\n", idx , testScore[idx]);

   }
   return 0;
}
```

```
int pointer_arithmetic_array_variable(void)

{

   int  idx           = 0;
   int  testScore[3] = {4, 7, 1};
   int* iPtr          = testScore;

   for (idx = 0; idx < 3; idx++)
   {

    printf("The address of index %d of the array is
%p\n\n",idx, &iPtr[idx] );

    printf("The value at index %d of the array is
%d\n\n",idx, iPtr[idx] );

    }
   return 0;
}
```


D:\Docs\2010 ITTS\2010 uestc\2010 lecture_C\cp_cases\cp_case_power.exe
The address of index 0 of the array is 0022FF40
The value at index 0 of the array is 4
The address of index 1 of the array is 0022FF44
The value at index 1 of the array is 7
The address of index 2 of the array is 0022FF48
The value at index 2 of the array is 1
F30
F34
F38
The value at index 2 of the array is 1
%d


D:\Docs\2010 ITTS\2010 uestc\2010 lecture_C\cp_cases\cp_case_power.exe
The address of index 0 of the array is 0022FF40
The value at index 0 of the array is 00000004
The address of index 1 of the array is 0022FF44
The value at index 1 of the array is 00000007
The address of index 2 of the array is 0022FF48
The value at index 2 of the array is 00000001

%p

Pointer Arithmetic

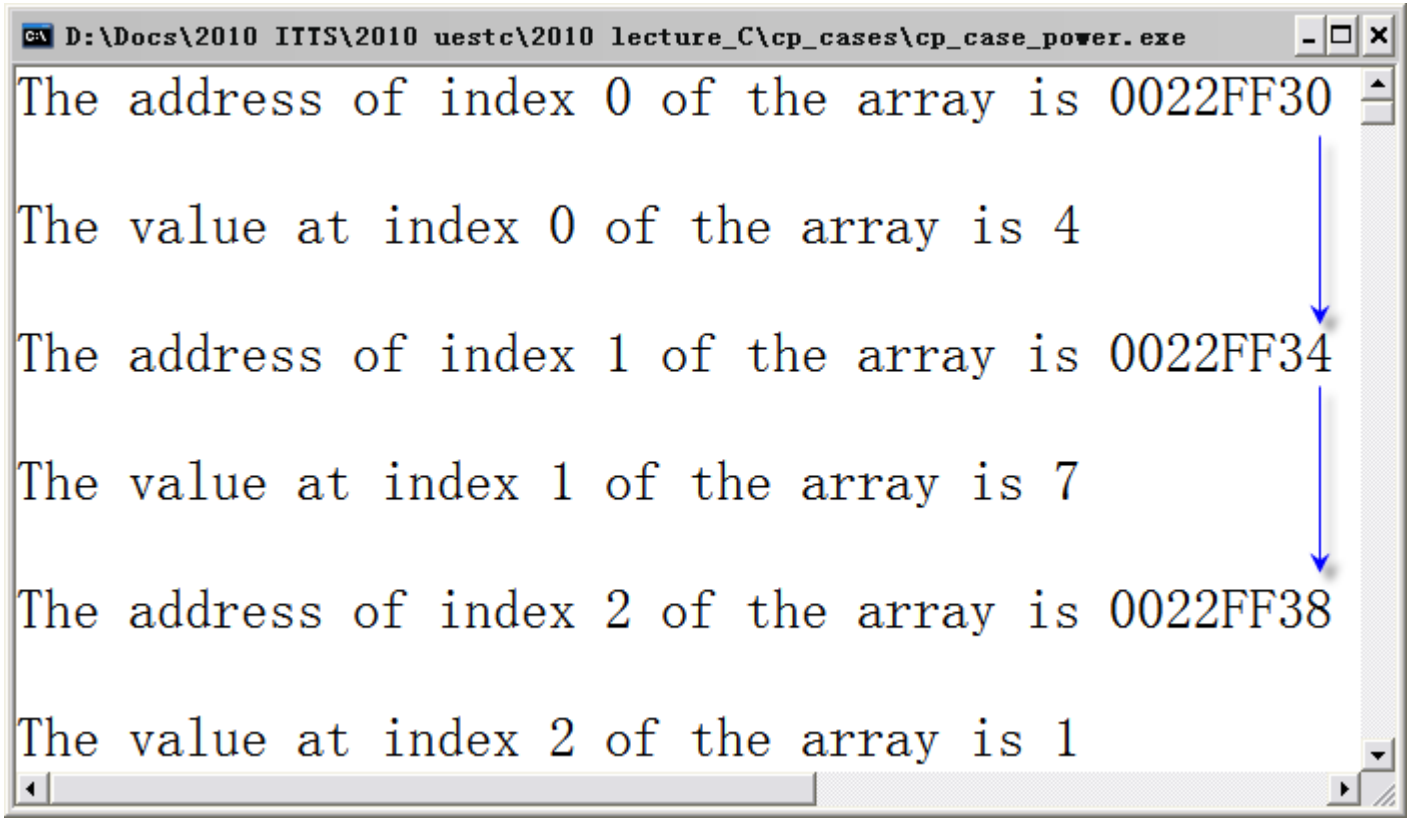
Case 2 : Incrementing/Decrementing a Pointer

iPtr++ / iPtr--

An important reason for declaring a **variable** pointer so it points to the same address as the array name is so the variable pointer can be **incremented**, unlike the array name which **cannot** be incremented because it is a **constant** pointer.

The following program increments the variable pointer to access each succeeding element of the array:

```
int pointer_arithmetic_array_incrementing(void)

{
   int  idx           = 0;
   int  testScore[3] = {4, 7, 1};
   int* iPtr          = testScore;

   for (idx = 0; idx < 3; idx++, iPtr++)
   {

    printf("The address of index %d of the array is
%p\n\n", idx, iPtr);

    printf("The value at index %d of the array is
%d\n\n", idx, *iPtr);

    }
   return 0;
}
```


D:\Docs\2010 ITTS\2010 uestc\2010 lecture_C\cp_cases\cp_case_power.exe
The address of index 0 of the array is 0022FF30
The value at index 0 of the array is 4
The address of index 1 of the array is 0022FF34
The value at index 1 of the array is 7
The address of index 2 of the array is 0022FF38
The value at index 2 of the array is 1

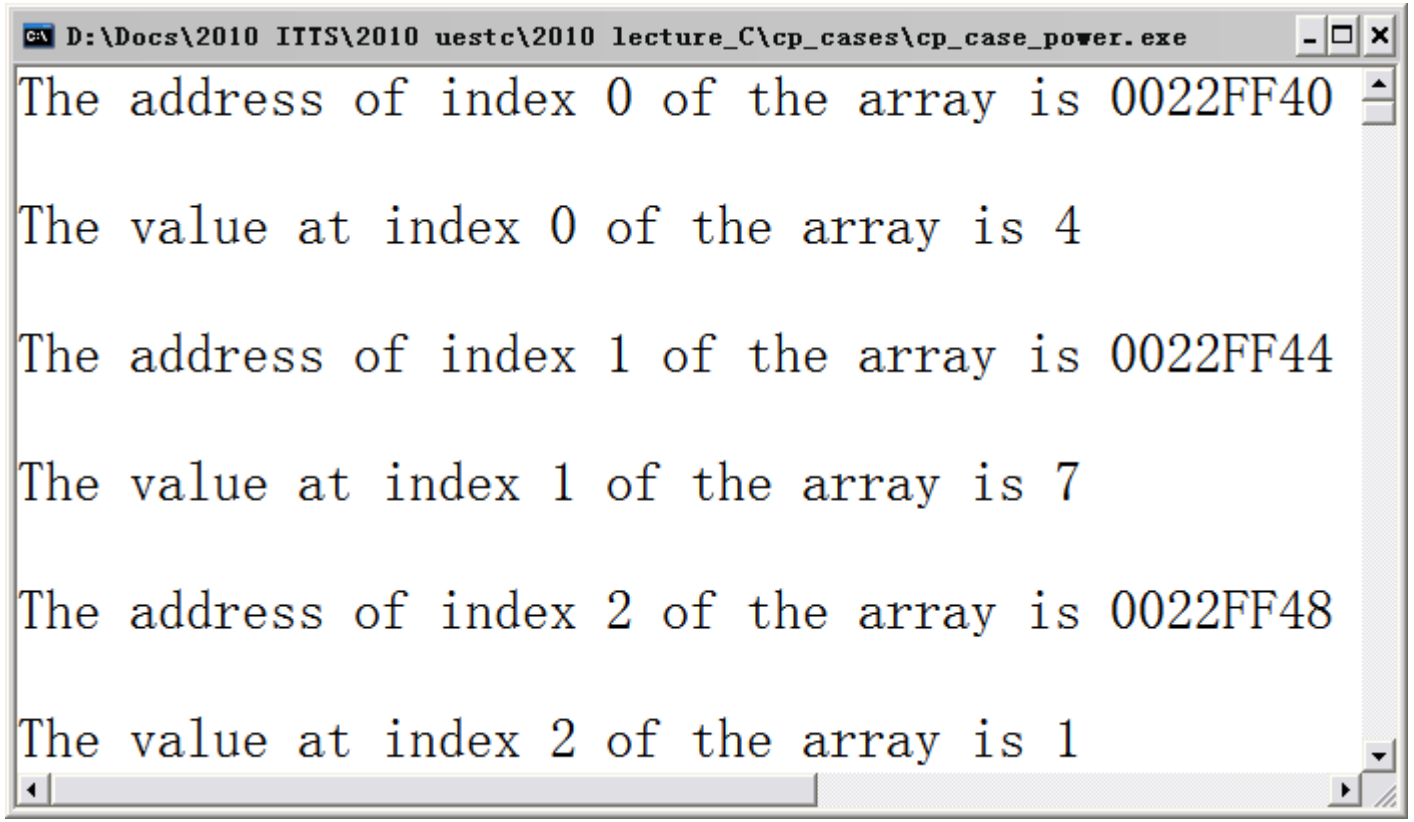
Pointer Arithmetic

Case 3 : Comparing Addresses

Addresses can be **compared** like any other value.

The following program modifies the previous one by incrementing the variable pointer so long as the address to which it points is either **less than or equal to** the address of the last element of the array, which is *&testScore[MAX - 1]:*

```
int pointer_arithmetic_array_comparing(void)
{

   int  testScore[3] = {4, 7, 1};
   int* iPtr         = testScore;
   int  idx          = 0;

   while ( iPtr <= &testScore[3-1] )
   {
    printf("The address of index %d of the array is
%p\n\n", idx, iPtr);

    printf("The value at index %d of the array is
%d\n\n", idx, *iPtr);

      iPtr++; /*iPtr = iPtr + 1*/
      idx++;
   }

   return 0;  }
```


D:\Docs\2010 ITTS\2010 uestc\2010 lecture_C\cp_cases\cp_case_power.exe
The address of index 0 of the array is 0022FF40
The value at index 0 of the array is 4
The address of index 1 of the array is 0022FF44
The value at index 1 of the array is 7
The address of index 2 of the array is 0022FF48
The value at index 2 of the array is 1

Pointer Arithmetic

iPtr **+ *1*** is **not** the base address **plus 1**, but instead is the base **address + 4.**
The same is true of *testScore* + *1.*

Consequently, the value at the **second element** of the array can be expressed one of **four** ways:

```
  /* start from [0] */

  testScore[1];      /* constant */

  *(testScore + 1);  /* constant */

  iPtr[1];           /* variable */

  *(iPtr + 1);       /* variable */
```

Pointer Arithmetic

The key statement is

```
char* str = "Jeff Kent";
```

This statement is almost the same as:

```
char str[] = "Jeff Kent";
```

In both statements, *str* is a character pointer, and **implicit** array **sizing** is used.

The difference is that *str* in the first statement (*char* str*) is a **variable** pointer

whereas *str* in the second statement (*char str[]*) is a **constant** pointer.

Contents

Pointers as Function Arguments

Pointers may be passed **as function arguments.** Pointer **notation(*)** usually is used to note that an argument is a pointer. However, if the pointer argument is the name of an array, subscript notation **alternatively** may be used.

If you want to:

1) variable *data* as output, use *

2) array as output, use **

3) IF *() as output, use higher level **()

4) As input, use *data* or *data pointer* are both acceptable

Pointers as Function Arguments

a sorting routine might **exchange** two out-of-order arguments with a function called **swap**.

Function name: ***swap()***
Inputs: ***xdata, ydata***
Outputs: ***ydata, xdata***

```
void swap( int xdata, int ydata )  /* WRONG */
  {
     /*initialise a temporary variable*/
      int temp = 0;

      temp  = xdata;
      xdata = ydata ;
      ydata = temp;
  }
```

Pointers as Function Arguments

/* Recommended*/

```
int swap( int* xdata,/*I/O*/
          int* ydata /*I/O*/ )
  {
     /*initialise a temporary variable*/
      int temp  = 0;
      int error = 0;

       temp   = *xdata;
      *xdata  = *ydata ;
      *ydata  =  temp;

      …  /* do something to check error*/
       return error;
  }
```

Pointers as Function Arguments

I – Input

O – Output

I/O – Input and Output

OF – Output and Need to be freed later later

I/OF – Input, Output and Need to freed later

Contents

Dynamic Memory Allocation

when declaring an **array**, the **size** declarator must be either a literal or a constant, and may **not** be a **variable.**

The compiler needs to know **exactly** how much memory to allocate.

However, if a variable is the size declarator, the compiler **does not** know how much memory to allocate, because a **variable's value** may **change.**

The value of the variable used as the size declarator is **not** even known **until runtime.**

If the length of the variable is **unknown**,

To accomplish this, you need to declare the array using **dynamic memory allocation.**

Dynamic Memory Allocation

Length Known

```
/*the length of array will always be the exact one
during all calculating period*/

#define MAX_STRING_LEN 255

char    file_name [MAX_STRING_LEN+1] = '\0';

double  data [256] = {0.0};
```

```
#define N_ELEMENTS(array) (sizeof(array)/sizeof(array[0]))
```

Dynamic Memory Allocation

Length Unknown

Case 1

```
int  n_bodies  = 0 ;

/* = int data , will be generated by another function
during calculating ,dynamically*/

tag_t   *bodies = NULL ;
/*step here, n_bodies have int value now, but it may has
different int value during calculating*/
n_bodies  = 100;

bodies = (tag_t *)malloc( n_bodies * sizeof(tag_t));
...
free(bodies);

bodies = NULL ;
```

Dynamic Memory Allocation

Length Unknown

Case 2

```
double (*bounding_boxes)[6] = …;
int    n_sheets  = …

/* = int data , will be generated by another function
during calculating ,dynamically*/

bounding_boxes = (double (*)[6])malloc( n_sheets *
sizeof(double[6]));

…

free(bounding_boxes);
```

Contents

Returning Pointers from Functions

if you **dynamically allocate memory** inside a function using a local pointer by ***malloc()***, then when the function terminates, the pointer will be destroyed but the memory will remain, it need to be handled by ***free()***

```
char* ask_the_name (void)
{

char name[80];

scanf("%s", name);

…

return name;

}
```

If a few pointers are taken as the outputs, see I/O

FAQ Contents

FAQ1: What is a pointer?

FAQ2: Name a C task that requires a pointer to be performed.

FAQ3: What is the difference between declaring an integer variable and declaring an integer pointer variable?

FAQ4: What is the meaning of the data type in the declaration of a pointer?

>>> more

FAQ

FAQ1: What is a pointer?

A pointer *is a variable or constant whose value is* ***the address*** *of another variable or constant.*

FAQ2: Name a C task that requires a pointer to be performed.

Dynamic memory allocation *requires a pointer to be performed.*

FAQ3: What is the difference between declaring an integer variable and declaring an integer pointer variable?

The only difference between declaring an integer variable and an integer pointer variable is that the pointer variable declaration includes an ***asterisk****, which either follows the* ***data type*** *or precedes the* ***variable name****.*

FAQ

FAQ4: What is the meaning of the **data type** in the declaration of a pointer?

*The data type in the declaration of a pointer refers to the **data type** of another variable (or constant) whose memory **address** is the value of the pointer.*

FAQ5: What is the meaning and **purpose** of **NULL**?

NULL** is a constant defined in several standard libraries. You assign a pointer NULL if it is **too early** in your code to know which address to assign to the pointer. The value of NULL, the memory address 0, signals that the pointer is not intended to point to an accessible memory location. - **initialisation

FAQ

FAQ6: What **operator** do you use to assign a pointer the address of another variable or constant?

*use the **address operator** to assign a pointer the address of another variable or constant*.

FAQ7: What is the purpose of the **indirection operator**?

*The purpose of the indirection operator is to **obtain** the value of the variable or constant to which the pointer points. This operation is said to dereference the pointer.*

FAQ

FAQ8: May a pointer point to **different memory addresses** at different times in the program?

*A pointer may point to different memory addresses at different times in the program if the pointer is declared as a **variable** instead of as a constant.*

FAQ9: May more than one pointer point to the same memory address?

***Yes**. More than one pointer may point to the same memory address.*

Quiz

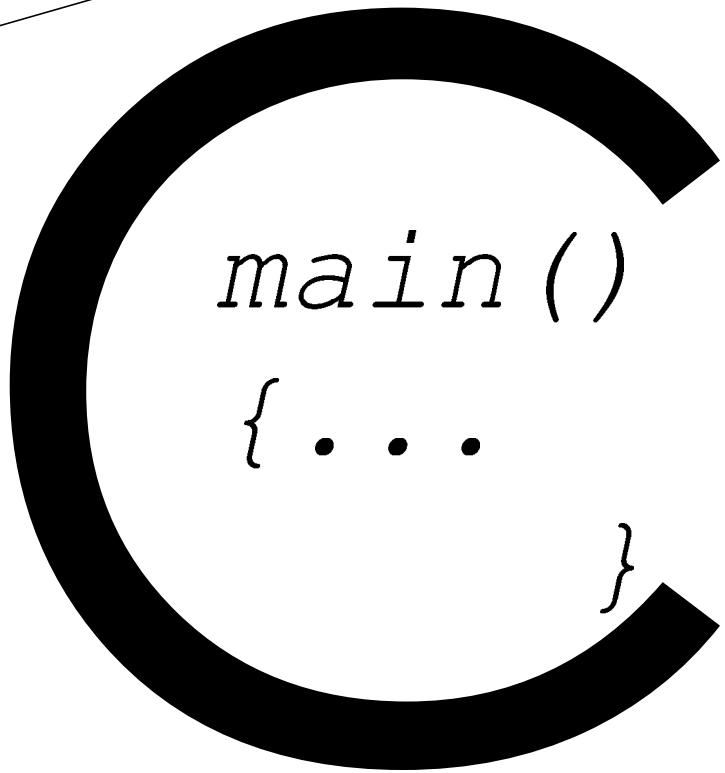
FAQ10: What is the effect of **incrementing** a **pointer variable**?

*Incrementing a pointer variable **increases** its value by the number of **bytes** of **its data type**.*

FAQ11: What are the purposes of the **new** and **delete** operators? (C++)

*The purpose of the **new** operator is to dynamically allocate memory. The purpose of the **delete** operator is to deallocate dynamically created memory.*

Chen , Yi
leo.chen.yi@live.co.uk
30-Jul-2010



Pointer
– the memory address

End

BETA 1.0.0.1


[卷珠帘.Scottish早春]
单衣薄袖拂垂柳。
飞花桥头，
初寒早春后。
杏花微雨黄藤酒，
尘缘往事牵谁手？
泪湿冷枕烛影瘦。
雪影初叠，
梦中红颜秀。
长烟归棹轻孤舟。
琵琶丝语泪痕旧。
2014-Feb-26, 词牌名"卷珠帘"是"蝶恋花"的别名，霍尊的歌曲《卷珠帘》，并不是按词牌来写词的。内容参考:[蝶恋花.余陵往事.Met You There] 和 [南乡子. Scottish Spring, Highland Mountains, Mount Hua]
LEO