POINTERS

Point to here, point to there, point to that, point to this, and point to nothing!
well, they are just memory addresses!!??

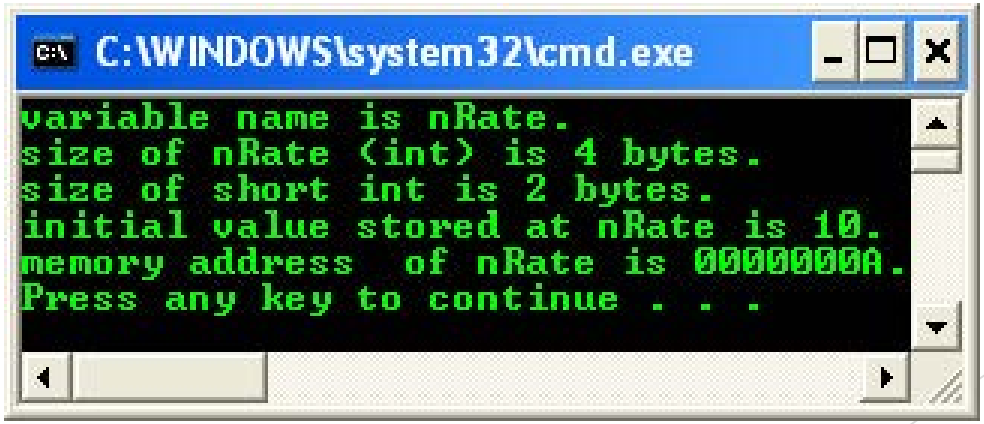
POINTERS

- When declaring a variable, the compiler sets aside memory storage with a unique address to store that variable.
- The compiler associates that address with the variable's name.
- When the program uses the variable name, it automatically accesses a proper memory location.
- No need to concern which address.
- But we can manipulate the memory address by using pointers.
- Let say we declare a variable named `nRate` of type integer and assign an initial value as follows,

```
int   nRate = 10;
```

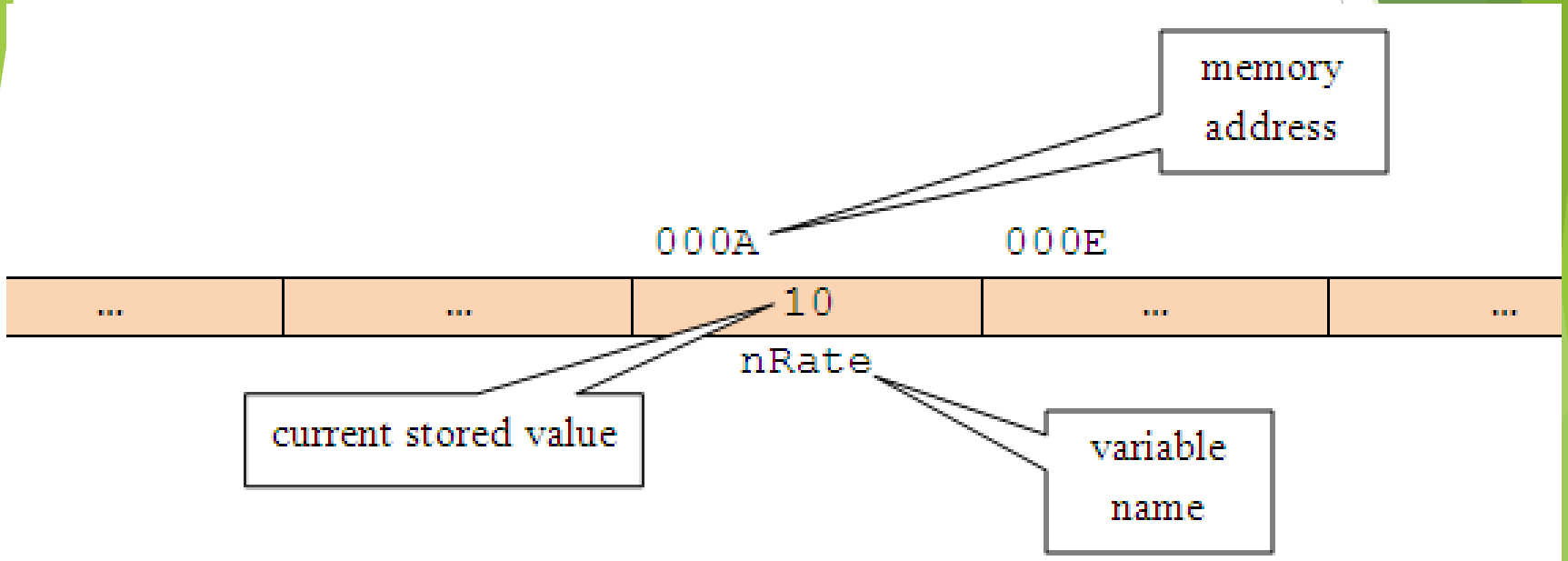
POINTERS

```c
#include <stdio.h>
void main(void)
{
        int nRate = 10;
        printf("variable name is nRate.\n");
        // or sizeof(int)
        printf("size of nRate (int) is %d bytes.\n", sizeof(nRate));
        printf("size of short int is %d bytes.\n", sizeof(short int));
        printf("initial value stored at nRate is %d.\n", nRate);
        printf("memory address  of nRate is %p.\n", nRate);
}
```



POINTERS

- The variable is stored at specific memory address and can be depicted as follows,

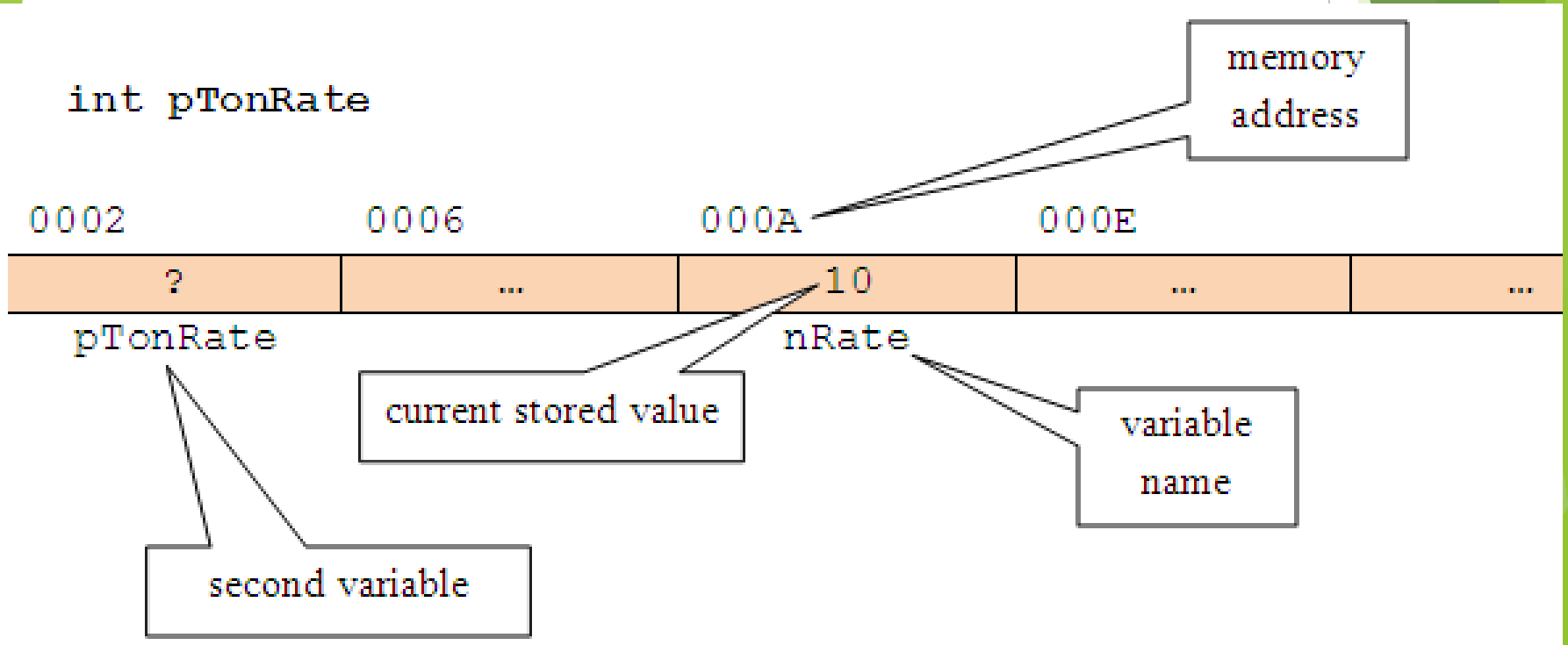


POINTERS

- The variable scope is local to the `main()` function.
- The memory allocation is from the `main()` function's stack.
- The `nRate` memory address (or any other variable) is a number, so can be treated like any other number in C.
- Using %p, the address is in hexadecimal format.
- Hence, if a variable's memory address is 'known', we can create a second variable for storing the memory address.
- From the previous example, let declare another variable, named `pTonRate` to hold the address of `nRate` variable.

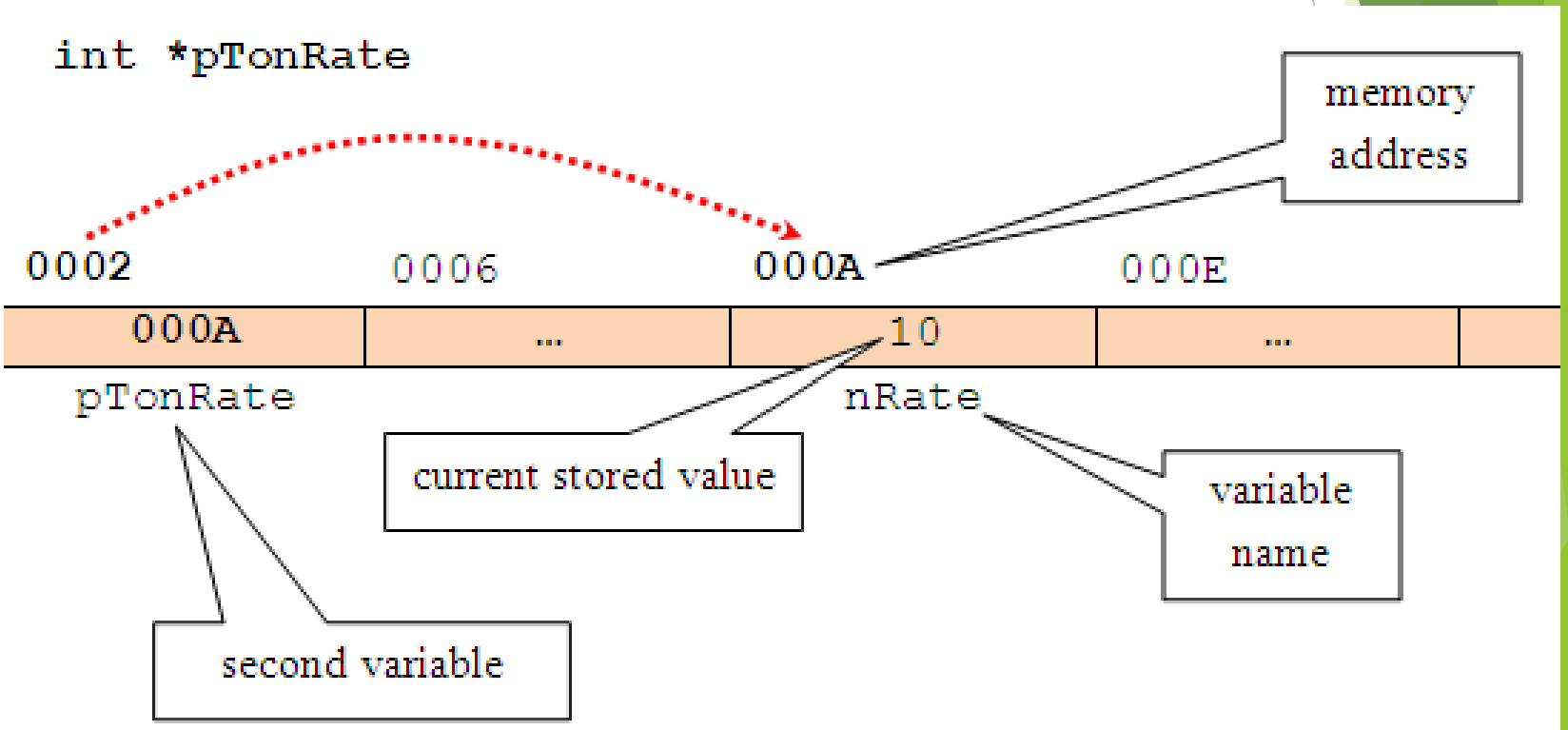
POINTERS

- At the beginning, `pTonRate` is uninitialized.
- So, storage has been allocated for `pTonRate`, but its initial value is undetermined, as shown below,



POINTERS

- Let store the `nRate`'s memory address, in `pTonRate` variable.
- So, `pTonRate` now holds the `nRate`'s memory address, where the actual data (10) is stored.
- Finally, in C vocabulary, `pTonRate` is pointing to `nRate` or is said a pointer to `nRate` and can be illustrated as follows,



POINTERS

- So, the pointer variable declaration becomes something like this,

```
int   *pTonRate;
```

- In other word, the `pTonRate` variable holds the memory address of `nRate` and is therefore a pointer to `nRate`.
- The asterisk (*) is used to show that is it the pointer variable instead of normal variable.

Definition: A pointer is a variable that holds memory address of another variable, where, the actual data is stored.

POINTERS

- By using pointers, it is an efficient way of accessing and manipulating data.
- Very useful for dynamic memory management, as we know memory, registers and cache are scarce resource in computer system.
- Every computer system, there are fixed and limited amount of memory for temporary data storage and processing.
- There are loading and unloading data into and from, moving data from and to different locations of the memory, including the cache and the processors' registers for temporary data storage and processing.
- The loading and unloading need an efficient memory management.
- Some of the memory portion also used for shared data to further optimizes the utilization.
- This shared data access (read/write) depends heavily on the pointers manipulation.
- Without moving/loading/unloading data to/from different locations in memory, we can use pointers to point to different locations.

POINTERS

Pointers and Simple Variables

- A pointer is a numeric variable and like other variables, must be declared and initialized before it can be used.
- The following is a general form for declaring a pointer variable,

```
type_of_stored_data    * pointer_variable_name;
```

- For example,

```
char*    chName;
int   *    nTypeOfCar;
float    *fValue;
```

- *type_of_stored_data* is any valid pointer base type such as `char`, `int`, `float` or other valid C derived types such as array and `struct`.
- It indicates the type of the variable's data to which the pointer is pointed to.
- The *pointer_variable_name* follows the same rules as other C variable naming convention and must be unique.

POINTERS

- From the pointer declaration examples, in the first line, the declaration tells us that `chName` is a pointer to a variable of type `char`.
- The asterisk (*) is called indirection operator, and it indicates that `chName` is a pointer to type `char` and not a normal variable of type `char`.
- Note the position of the *, it is valid for all the three positions.
- Pointers can be declared along with non pointer variables as shown below,

```
// chFirstName and chSecondName both are pointers to
// type char while chInitial is normal char type variable
char    *chFirstName, *chSecondName, chInitial;
// fValue is a pointer to type float, and fPercent
// is an ordinary float type variable.
float    *fValue, fPercent;
```

POINTERS

Initializing Pointers

- Once a pointer is declared, we must initialize it.
- This makes the pointer pointing to something.
- Don’t make it point to nothing; it is dangerous.
- Uninitialized pointers will not cause a compiler error, but using an uninitialized pointer could result in unpredictable and potentially disastrous outcomes.
- Until pointer holds an address of a variable, it isn’t useful.
- C uses two pointer operators,

1. Indirection operator (*) – asterisk symbol, has been explained previously.
2. Address-of-operator (&) – ampersand symbol, means return the address of…

- When & operator is placed before the name of a variable, it will returns the memory address of the variable instead of stored value.

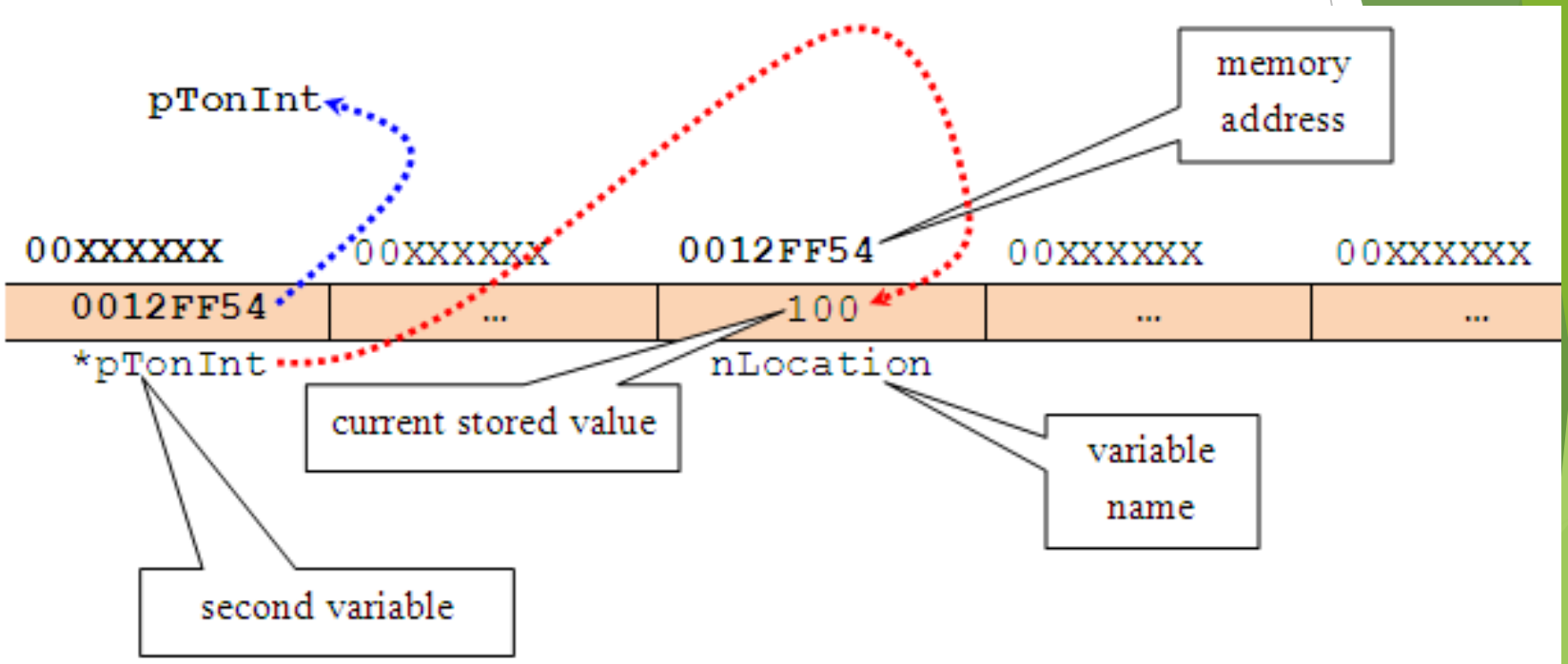
POINTERS

- Program example: initializing a pointer

- In this example, pointer variable, `pToInt` receives the address of `nLocation` or
- The memory address of the `nLocation` variable is assigned to `pToInt` pointer variable.

POINTERS

- Graphically this situation can be illustrated as shown below.



- The * operator is a complement of & operator.
- * means returns the current value of the variable pointed to.

POINTERS

- From the previous example, if we declare another variable of type `int`,

```
int anotherVar;
int  * pToInt;
pToInt = &nLocation;
nLocation = 100;
```

- Then, statement,

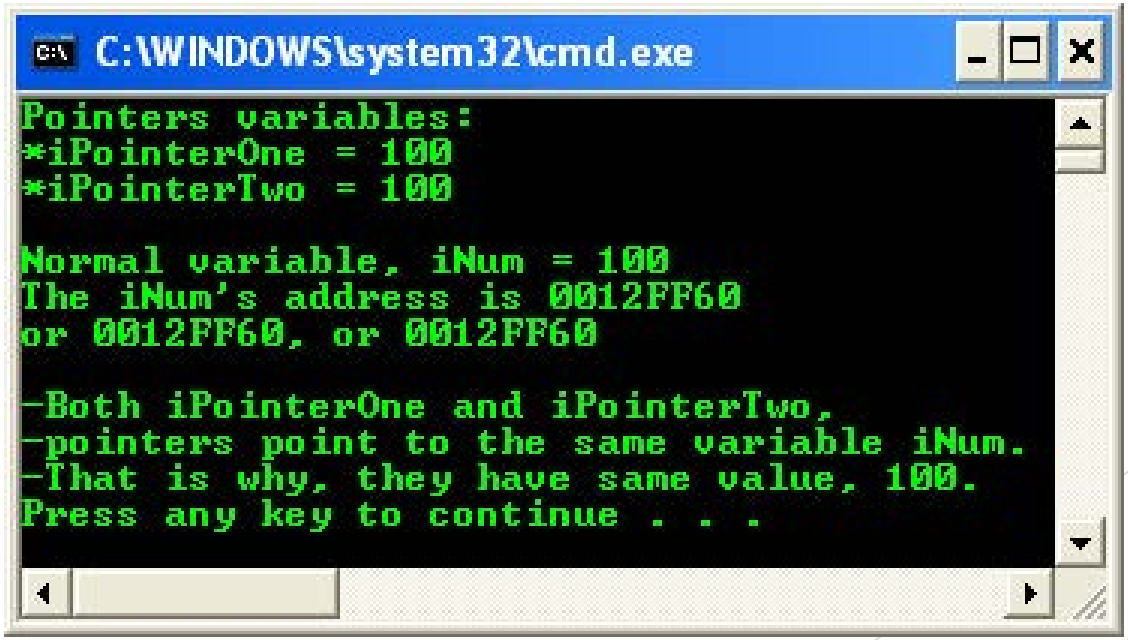
```
anotherVar = *pToInt;
```

- will place the actual current data value stored in variable `nLocation` into variable `anotherVar`.
- Which means `anotherVar` was assigned the actual current data value stored at the memory address held by `pToInt`.
- The `*` operator appears before a pointer variable in only two places:

1. When declaring a pointer variable.
2. When de-referencing a pointer variable (to show the data it's pointed to).

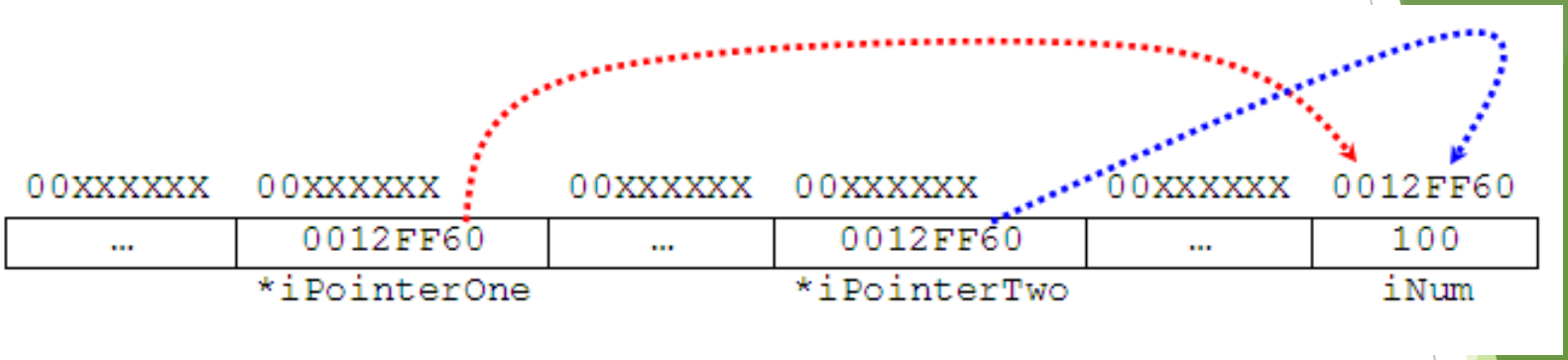
POINTERS

- Only the addition and subtraction operations are permitted in pointer's expression.
- As with any variable, a pointer may be used on the right hand side of an assignment statement to assign its value to another pointer as shown in the following example.
- Program example: using pointers



POINTERS

- The situation can be illustrated as shown below. The memory address should be different for different run/system.



- Take note the difference between memory address and the actual data value, which stored at the memory address.
- Do not worry about the address, which are determined and provided by the system.
- From the previous example, we can:
  1. Access the variable's content by using the variable name (`iNum`) and is called direct access.
  2. Access the variable's content by using a pointer to the variable (`*iPointerOne` or `*iPointerTwo`) and is called indirect access or indirection.

POINTERS

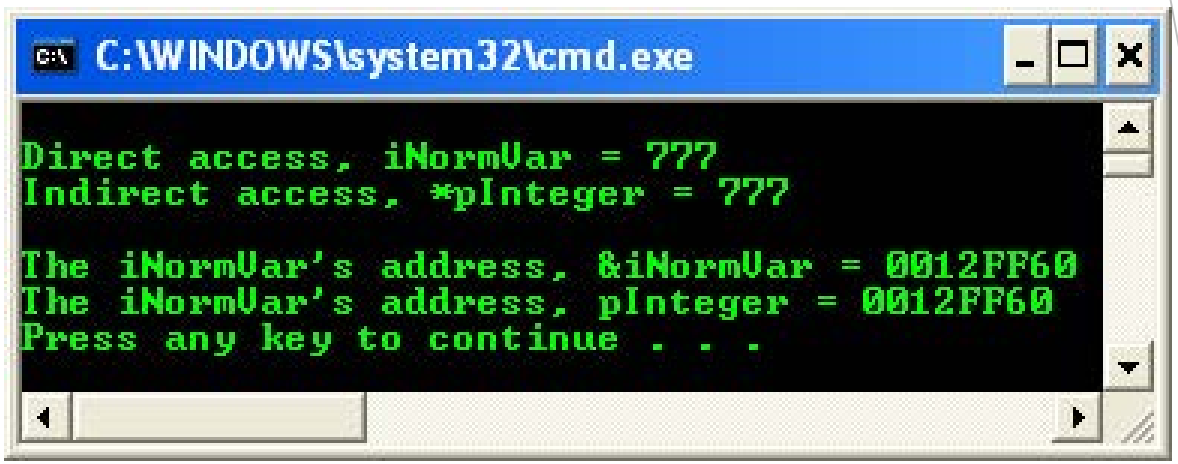
- As a conclusion, if a pointer named `pPointerVar` of type `int` has been initialized to point to a variable named `iNumVar`, the following are true,

```
// declare a pointer variable named pPointerVar, where the
// data stored pointed to by pPointerVar is int type
int * pPointerVar;
// assign the iNumVar's address to pPointerVar pointer variable
pPointerVar = &iNumVar;
```

- `* pPointerVar` and `iNumVar` both refer to the contents of `iNumVar` (that is, whatever data value stored there).
- `pPointerVar` and `&iNumVar` refer to the address of `iNumVar` (the `pPointerVar` only hold the address of variable `iNumVar` not the actual data value).
- So, a pointer name without the indirection operator (*) accesses the pointer value itself, which is of course, the address of the variable pointed to.

POINTERS

- Program example: using pointers operator



- The address displayed for variable `iNormVar` may be different on your system because different computer will have different specification.

POINTERS

- For pointer arithmetic operation, only two arithmetic operations, that is addition and subtraction available. For example,

```
int    iAge = 35;
```

- Hence, C reserved storage for the variable `iAge` and store the value 35 in it.
- Let say, C has stored this value at the memory address `1000`, then we declare a pointer variable named `pPointerToAge` that point to the `iAge` variable.
- Then, after the following expressions have been executed,

```
int * pPointerToAge;
pPointerToAge = &iAge;
pPointerToAge++;
```

- The content of the pointer then becomes `1004` instead of `1001` (integer value takes 4 bytes so C add 4 to the pointer).

POINTERS

- Different variable types occupy different amount of memory, implementation dependent, following C standard or implementation extension and whether 16, 32 or 64 bits system.
- Each time the pointer is incremented, it points to the next integer and similarly, when a pointer is decremented, it points to the previous integer.
- Each individual byte of memory has its own address; so multi-byte variable actually occupies several addresses.
- When pointers used to handle the addresses of multi-byte variables, the address of a variable is actually the address of the lowest byte it occupies.
- For example,

```
int     iNum = 12252; // 4 bytes
char    chVar = 90; // 1 byte
float   fVar = 1200.156004; // 4 bytes
```

POINTERS

- The pointer variables are pointing to the lowest byte of variable memory slots e.g: `iNum` occupies 4 bytes and `chVar` occupies 1 byte.
- Other pointer arithmetic operation is called differencing, which refers to subtracting two pointers.
- For example, two pointers that point to different elements of the same array can be subtracted to find out how far apart they are.
- Pointer arithmetic automatically scales the answer, so that it refers to the array elements.
- Thus, if `pPointer1` and `pPointer2` point to elements of an array (of any type), the following expression tells you how far apart the elements are,

```
pPointer1 – pPointer2;
```

POINTERS

- However `ptrdiff_t` type can be used to return the subtraction operation between two pointers.
- It is a base signed integer type of C/C++ language.
- The type's size is chosen so that it could store the maximum size of a theoretically possible array of any type.
- On a 32-bit system `ptrdiff_t` will take 32 bits, on a 64-bit one 64 bits.
- The `ptrdiff_t` type enable you to write well-portable code.
- `ptrdiff_t` (together with `wchar_t` and `size_t`) defined in stddef.h (cstddef for C++).

POINTERS

Pointers Comparison

- The comparison is valid only between pointers that point to the same array.
- The following relational operators work for pointers operation: ==, !=, >, <, >= and <=.
- A lower array element that is those having a smaller subscript, always have a lower address than the higher array elements.
- Thus if `pPointer1` and `pPointer2` pointing to the elements of the same array, the following comparison is TRUE,

```
pPointer1 < pPointer2
```

- If `pPointer1` points to an earlier member of the array than `pPointer2` does.
- Many arithmetic operations that can be performed with regular variables, such as multiplication and division, do not work with pointers and will generate errors in C.

POINTERS

- The following table summarizes pointer operations.

Operation	Description
1. Assignment (=)	You can assign a value to a pointer. The value should be an address with the address-of-operator (&) or from a pointer constant (array name)
2. Indirection (*)	The indirection operator (*) gives the value stored in the pointed to location.
3. Address of (&)	You can use the address-of operator to find the address of a pointer, so you can use pointers to pointers.
4. Incrementing	You can add an integer to a pointer to point to a different memory location.
5. Differencing	You can subtract an integer from a pointer to point to a different memory location.
6. Comparison	Valid only with two pointers that point to the same array.

POINTERS

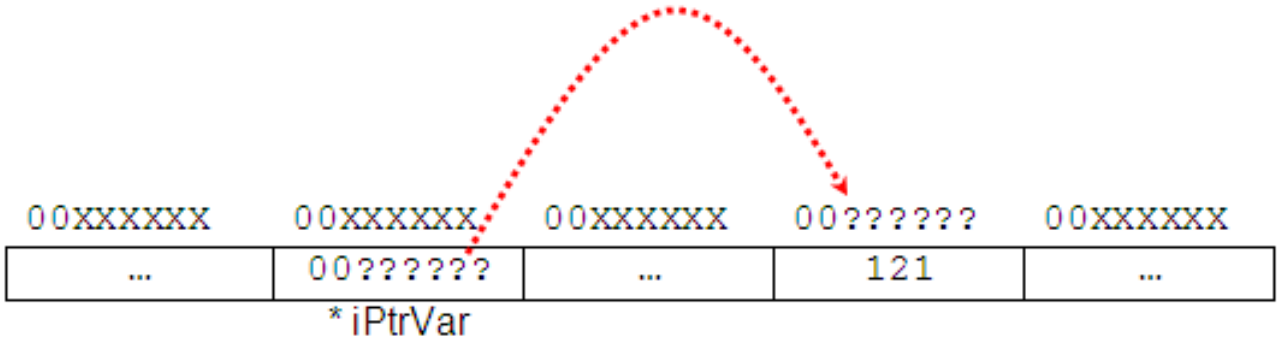
Uninitialized Pointers Issue

- Let say we have declared a pointer something like this,

```
int  *iPtrVar;
```

- This statement declares a pointer to type `int` but not yet initialized, so it doesn't point to anything known value (address).
- Then, consider the following pointer assignment statement,

```
// this is not an address that system allocate!
*iPtrVar = 121;
```



POINTERS

- This statement means the value of `121` is assigned to whatever address (`00??????`) `iPtrVar` is pointing to.
- That address can be anywhere in memory which may contain critical data.
- The `121` stored in that location may overwrite some important information, and the result can be anything from program error to a full system crash.
- We must initialize a pointer so that it point to something. Do not create a stray pointer.
- A better solution, we can assign NULL (`\0`) value during the initialization before using it, such as,

```
int  *iPtrVar = NULL; /* null pointer, pointing to something
   but nothing */
```

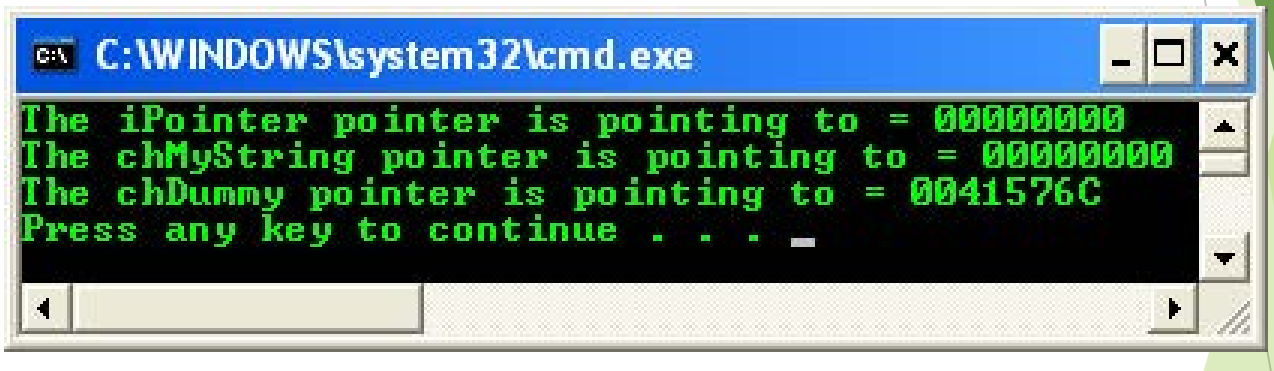
- For pointer that point to a string you may just point to an empty string for a dummy such as,

```
char    *chMyString = " ";
```

- The .NET C/C++ programming uses `nullptr` instead of `NULL`. The
- `NULL/nullptr` is a pointer that point to the `0x00000000` (32 bits system) memory address.

POINTERS

- Program example: NULL pointer and empty string



POINTERS

Arrays and Pointers

- A special relationship exists between pointers and arrays.
- An array name without brackets is a pointer to the array's first element.
- So, if a program declared an array,

```
char chName[10];
```

- `chName` (array's name) is the address of the first array element and is equivalent to the expression,

```
&chName[0]
```

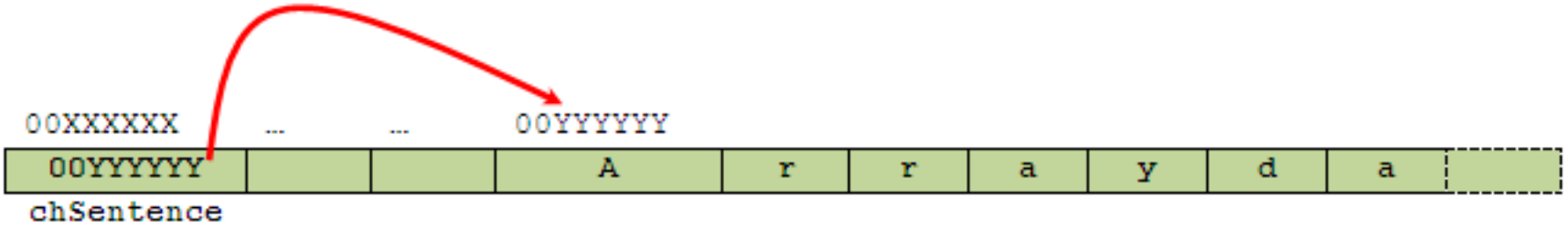
- Which is a reference to the address of the array's first element.
- `chName` equivalent to `&chName[0]`.
- The array's name is, therefore a pointer to the array's first element and therefore to the string if any.

POINTERS

- A pointer is a constant, which cannot be changed and remains fixed for the duration of program execution.
- Many string operations in C are typically performed by using pointers because strings tend to be accessed in a strictly sequential manner.
- Program example: pointers and array

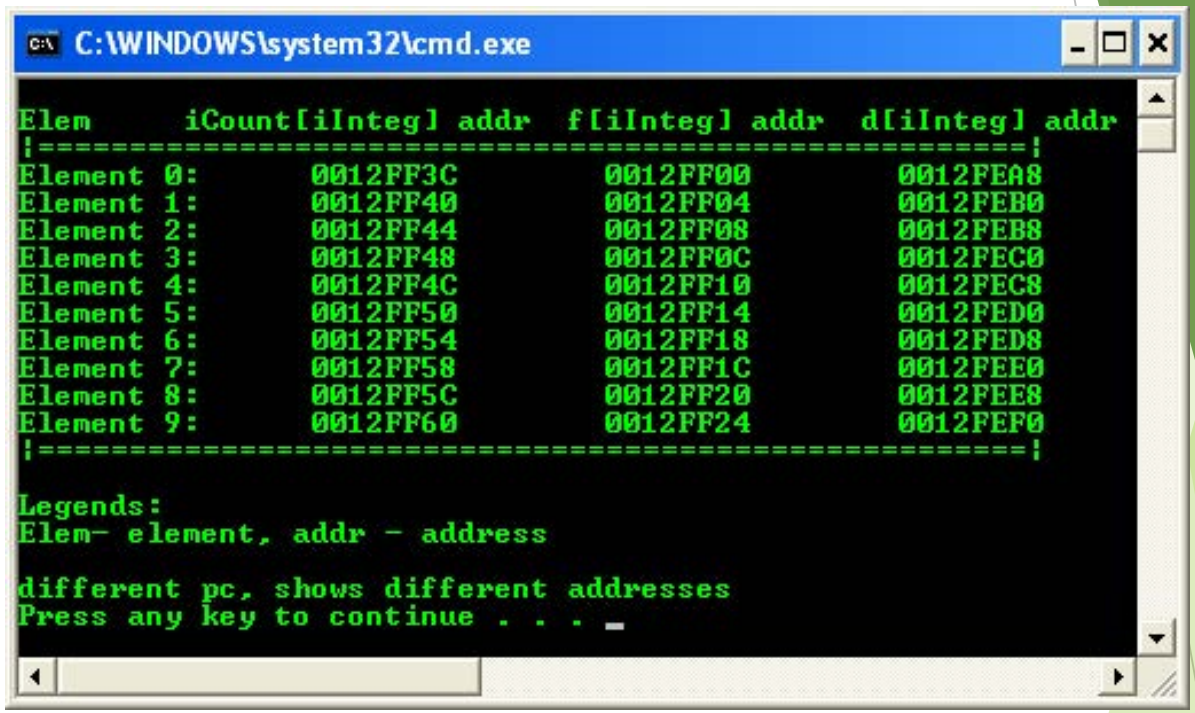
POINTERS

- Which can be illustrated as follows,



POINTERS

- Let take a look at the following example: pointers and array

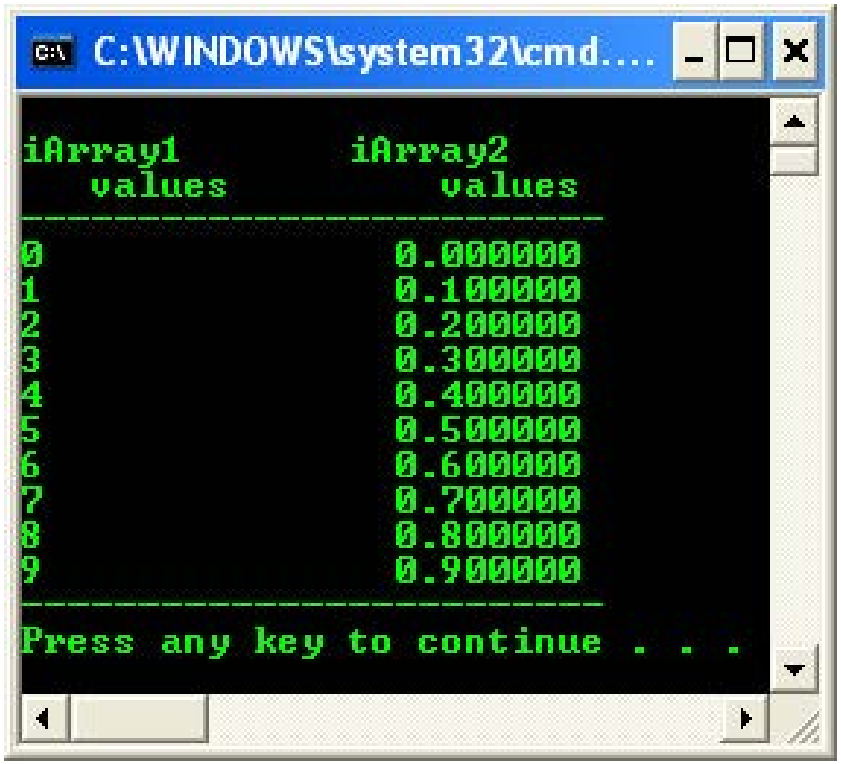


- Notice the difference between the element addresses.

`12FF40 – 12FF3C` = 4 bytes for `int`
`12FF04 – 12FF00` = 4 bytes `float`
`12FEB0 – 12FEA8` = 8 bytes `double`

POINTERS

- Try another program example: pointers and array access



POINTERS

- Let re-cap, if an array named `iListArray[ ]` is a declared array, the expression `* iListArray` is the array's first element, `*( iListArray + 1)` is the array's second element, and so on.
- Generally, the relationship is,

```
*(iListArray)  ==  iListArray[0]     // first element
*(iListArray + 1) == iListArray[1]   // second element
*(iListArray + 2) == iListArray[2]   // third element
...
...
*(iListArray + n) == iListArray[n]   // the nth element
```

- So, you can see the equivalence of array subscript notation and array pointer notation.

POINTERS

Arrays of Pointers and Function

- Pointers may be arrayed like any other data type.
- For example, a pointer array `iArrayPtr` of sized `20` is declared as,

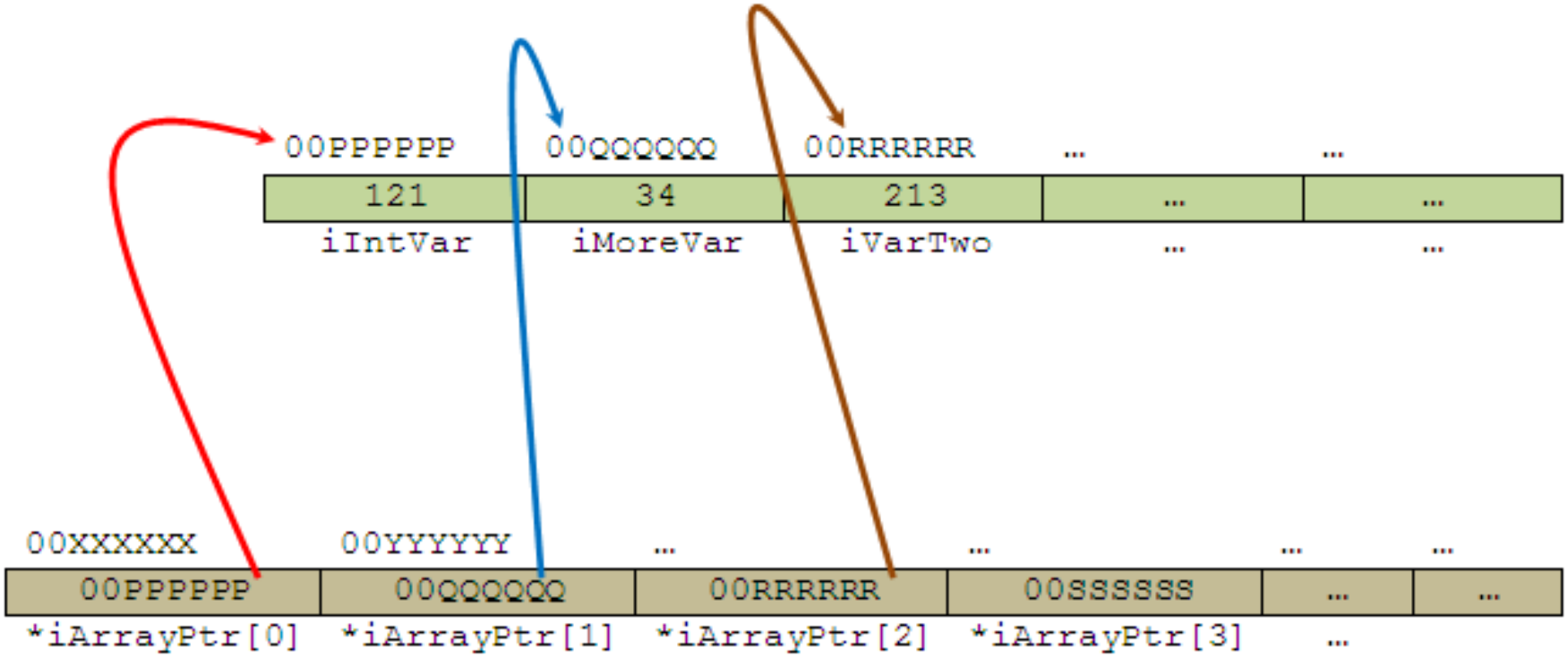
```
int    *iArrayPtr[20];
```

- To assign the address of an integer variable called `iIntVar` to the first element of the array, we could write something like this,

```
// assign the address of variable
// iIntVar to the first iArrayPtr element
iArrayPtr [0] = &iIntVar;
```

POINTERS

- Graphically can be depicted as follows,



POINTERS

- To find the value stored in `iIntVar`, we could write something like this,

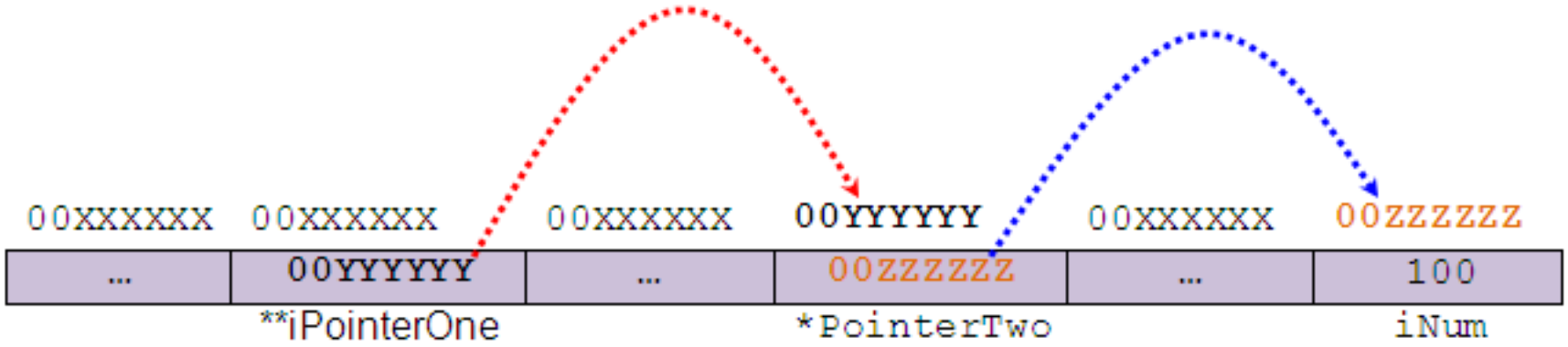
```
iArrayPtr[0]
```

- To pass an array of pointers to a function, we simply call the function with the array's name without any index/subscript, because this is an automatically a pointer to the first element of the array, as explained before.
- For example, to pass the array named `iArrayPtr` to `viewArrayFunc()` function, we use the following statement,

```
viewArrayFunc(iArrayPtr);
```

POINTERS

- Graphically, the construct of a pointer to pointer can be illustrated as shown below.
- `iPointerOne` is the first pointer, pointing to the second pointer, `iPointerTwo` and finally `iPointerTwo` is pointing to a normal variable `iNum` that hold integer `100`.



POINTERS

- In order to indirectly access the target value pointed to by a pointer to a pointer, the asterisk operator must be applied twice. For example,

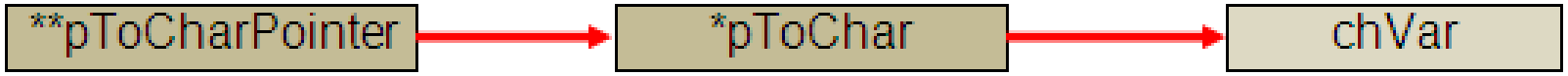
```
int   **iPointerOne;
```

- Tells compiler that `iPointerOne` is a pointer to a pointer of type integer.
- Pointer to pointer is rarely used but you will find it regularly in programs that accept argument(s) from command line.
- Consider the following declarations,

```
char  chVar;    /* a normal character variable */
char  *pToChar;  /* a pointer to a character */
char  **pToCharPointer; /* a pointer to a pointer to a
   character */
```

POINTERS

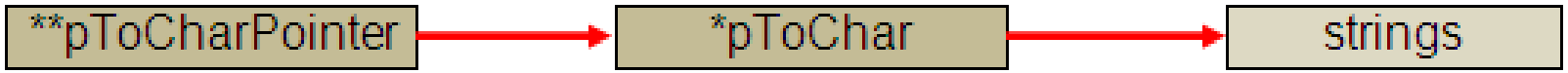
- If the variables are related as shown below,



- We can do some assignment like this,

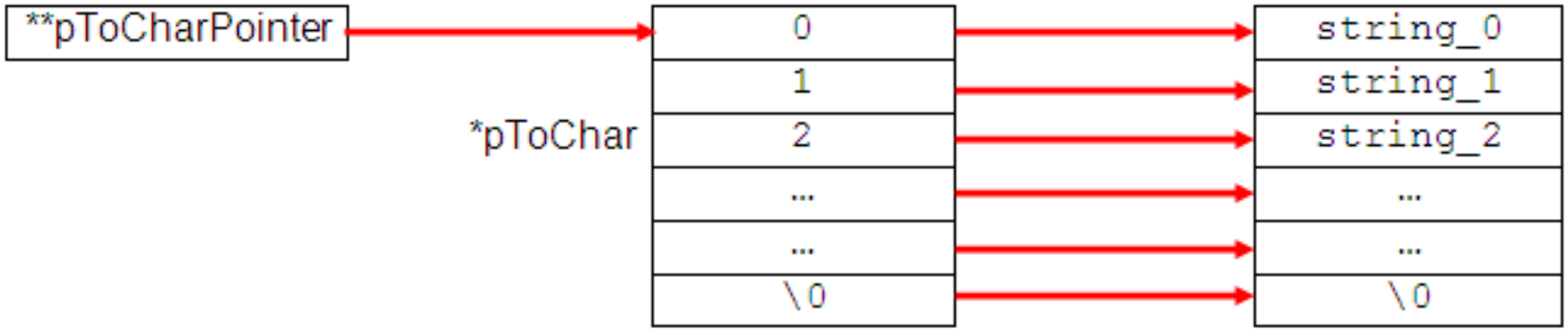
```
chVar = 'A';
pToChar = &chVar;
pToCharPointer = pToChar;
```

- Recall that `char *` (which is `*pToChar` in this case) refers to a `NULL` terminated string.
- Then, we can declare a pointer to a pointer to a string something like this,



POINTERS

- Taking this one stage further we can have several strings being pointed to by the integer pointers (instead of `char`) as shown below,



POINTERS

- Then, we can refer to the individual string by using `pToCharPointer[0]`, `pToCharPointer[1]`,.... and generally, this is identical to declaring,

```
char    * pToCharPointer[ ];  /* an array of pointer */
```

- Or from last figure,

```
char    ** pToCharPointer;
```

- Thus, programs that accept argument(s) through command line, the `main( )` parameter list is declared as follows,

```
int    main(int argc, char **argv)
```

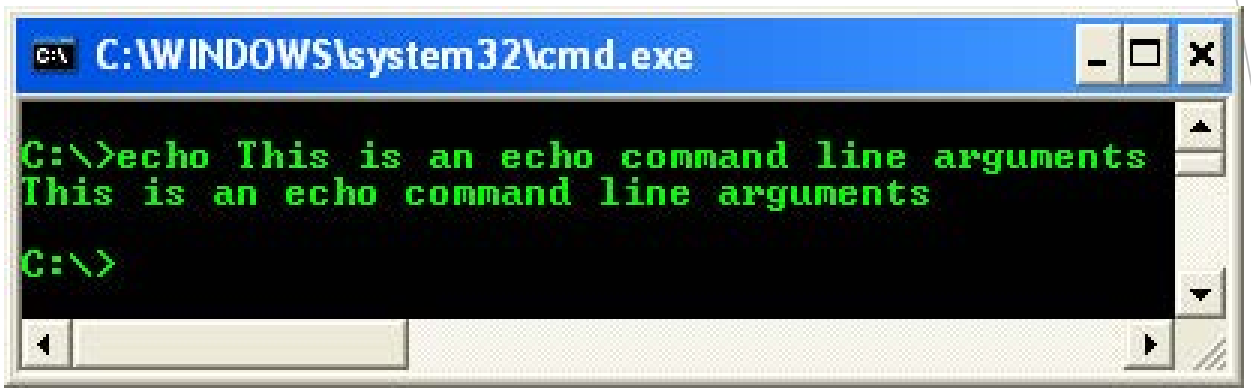
- Or something like this,

```
int    main(int argc, char *argv[ ])
```

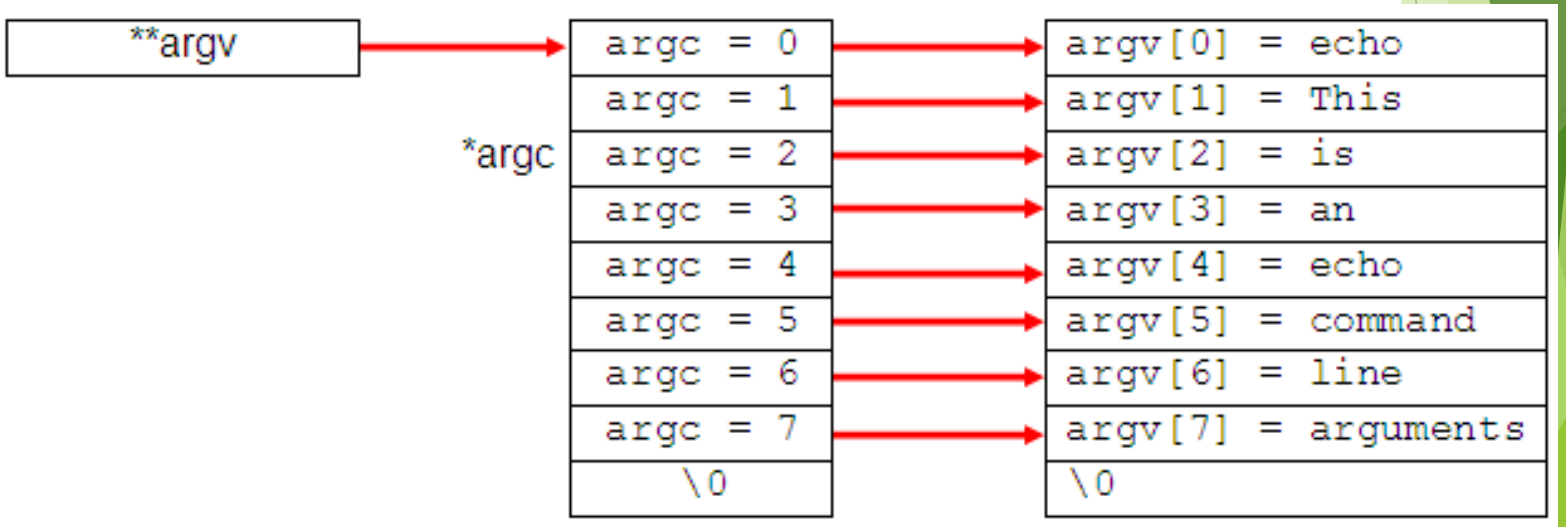
- Where the `argc` (argument counter) and `argv` (argument vector) are equivalent to `pToChar` and `pToCharPointer` respectively.

POINTERS

- For example, program that accepts command line argument(s) such as echo,



- This can be illustrated as,



POINTERS

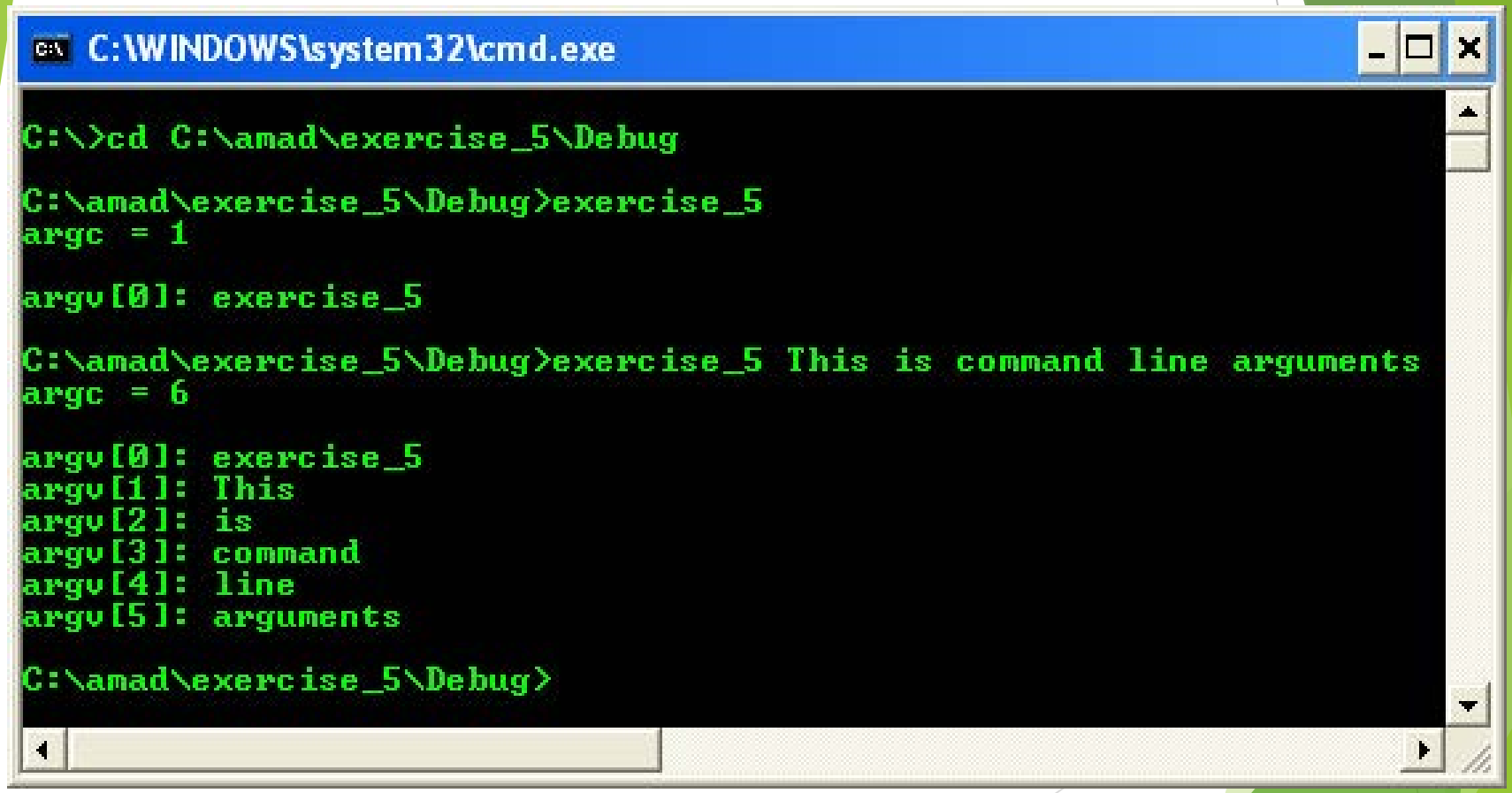
- So our `main()` function now has its own arguments.
- As a convention, these are the only arguments `main()` accepts.
- Notice that the `argv[0]` is an application name.
- `argc` (argument counter) is the number of arguments (unsigned integer), which we input including the program name.
- `argv` (argument vector) is an array of strings (`argv[0]`, `argv[1]`, `argv[2]`,...) holding each command line argument including the program name that is the first array element, argv[0].
- But some implementation will have another third parameter, envp/env, which is a pointer to an environment variable as shown below.

```
int     main(int argc, char *argv[ ], *envp[ ])
```

- Environment variable in Operating System is an array of strings.

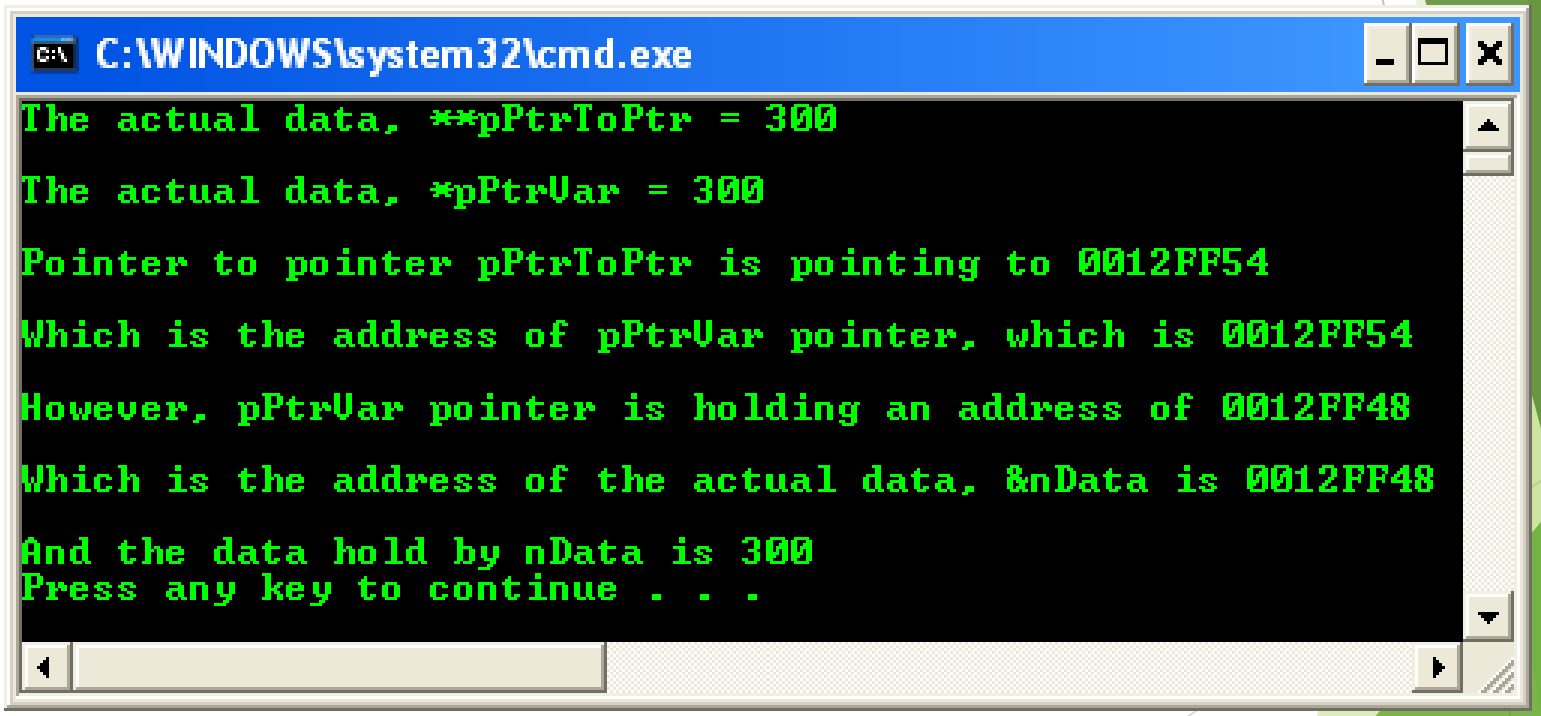
POINTERS

- Let try a program example (this program must be run at command prompt)



POINTERS

- Another pointer-to-pointer example



POINTERS

NULL Pointer Constant

- The `null` pointer constant is guaranteed not to point to any real object.
- You can assign it to any pointer variable since it has type `void *`.
- The preferred way to write a null pointer constant is with `NULL`.

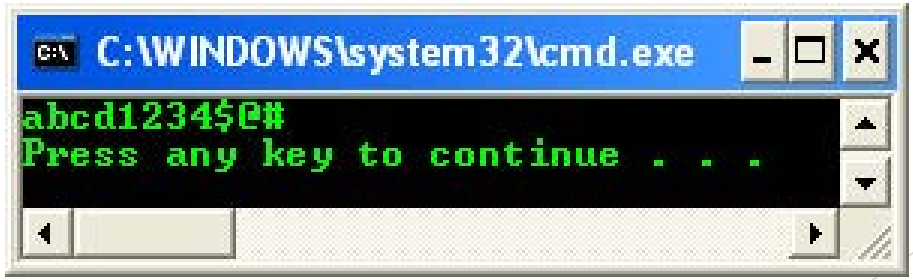
```
char * pNulPointer = NULL; or
char * pNulPointer = nullptr;
```

- `NULL` pointer when set, actually pointing to memory address `0x00000000` (32 bits).
- The computer systems have reserved this zero address for `NULL` pointer and that is why it cannot be used for other purposes.
- (in C++ .NET programming, `nullptr` is used as the keyword instead of `NULL`)
- Since C functions can only return one variable, it is common practice for those which return a pointer is set to `NULL`, can avoid the stray pointer.
- Set a pointer to `NULL` to indicate that it's no longer in use.

POINTERS

`void` Pointers

- There are times when you write a function but do not know the data type of the returned value.
- When this is the case, you can use a void pointer, a pointer of type `void`.
- Program example: void pointer



POINTERS

- Other pointer declarations that you may find and can make you confused are listed below.

Pointer declaration	Description
`int *x`	`x` is a pointer to `int` data type.
`int *x[10]`	`x` is an `array[10]` of pointer to `int` data type.
`int *(x[10])`	`x` is an `array[10]` of pointer to `int` data type.
`int **x`	`x` is a pointer to a pointer to an `int` data type – double pointers.
`int (*x)[10]`	`x` is a pointer to an `array[10]` of `int` data type.
`int *funct()`	`funct()` is a function returning an integer pointer.
`int (*funct)()`	`funct()` is a pointer to a function returning `int` data type – quite familiar constructs.
`int (*(*funct())[10])()`	`funct()` is a function returning pointer to an `array[10]` of pointers to functions returning `int`.
`int (*(*x[4])())[5]`	`x` is an `array[4]` of pointers to functions returning pointers to `array[5]` of `int`.

POINTERS

And something to remember!

`*`	- is a pointer to…
`[]`	- is an array of…
`()`	- is a function returning…
`&`	- is an address of…

END-of-C
POINTERS