WEEK 3 -POINTER

01 October 2024 11:18

The word pointer is also used to refer to just an adress;
Every pointer will be associated with a specific variable type, and it can be used only to point to variable of that type.

Why pointers instead? Because when you pass variable around you need to copy them while pointers being the adress when passed don't get copied, its more efficent memorywise

A pointer type void* can contain the adress of a data item of anytype [the adress name is the first byte of the type].

Type *pointrername;
You should initialize a declared pointet so it doesn't point to anything:

Int* pnumber = Null; (equal oìto zero for a pointer)

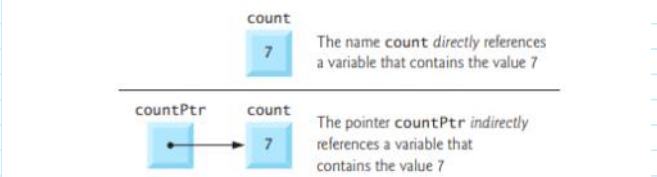
If you want to initialse your variable pnumber with the adress of a variable you have alredy declared:

Int number = 15; declare a variable stored an integer value 15
Int *pointer = &number; use a pointer to point to address of number

You can see the adress using %p
Printf("the adress is %p\n", &number)

POINTERS ARE VARIABLES WHOSE VARIABLE ARE MEMORY ADRESS. THEY CONTAIN THE ADRESS OF ANOTHER VARIABLE THAT CONTAIN A SPECIFIC VALUE, IT POINTS TO THAT VARIABLE

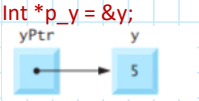
SO A VARIABLE DIRECTLY REFERENCE A VALUE
A POINTER INDERECTLY REFERENCE A VALUE



POINTER VARIABLE NAMING CONVECTION
-Ptr countPtr
p- pCount
p_ p_count

POINTEWR SHOULD BE INTIALIZED WHEN THEY ARE DEFINED OR THEY CAN BE ASSIGNED A VALUE.
A POINTER MAY BE INITIALIZED TO NULL OR 0
NULL is a symobolic constany with value zero

ADRESS(&) OPERATION
The urinart adress operation (&) return the adress
Int y = 5;
Int *p_y = &y;



Pointer rappresentation in memory:

The value following & must be a variable it cannot be applied to literal values like 27 or 41.4

The unary indirection operation (*)
Get the value of the object to which the pointer points.

printf("%d", *yPtr);
Using * in this manner is called dereferencing a pointer.
Dereferencing a pointer that has not been initialized with or assigned the address of another variable in memory is an error.

```c
    int a = 7;
    int *aPtr = &a; // set aPtr to the address of a

    printf("Address of a is %p\nValue of aPtr is %p\n\n", &a, aPtr);
    printf("Value of a is %d\nValue of *aPtr is %d\n\n", a, *aPtr);
    printf("Showing that * and & are complements of each other\n");
    printf("&*aPtr = %p\n*&aPtr = %p\n", &*aPtr, *&aPtr);
}
```

```
Address of a is 0x7fffe69386cc
Value of aPtr is 0x7fffe69386cc

Value of a is 7
Value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0x7fffe69386cc
*&aPtr = 0x7fffe69386cc
```

PASSING ARGUMENTS TO FUNCTION BY REFERENCE

TWO WAYS:

Pass by value: by defoult they are passed by value (other than array), function often need to modify value

```c
// fig07_02.c
// Cube a variable using pass-by-value.
#include <stdio.h>

int cubeByValue(int n); // prototype

int main(void) {
   int number = 5; // initialize number

   printf("The original value of number is %d", number);
   number = cubeByValue(number); // pass number by value to cubeByValue
   printf("\nThe new value of number is %d\n", number);
}
```

Fig. 7.2 | Cube a variable using pass-by-value. (Part 1 of 2.)


```c
// calculate and return cube of integer argument
int cubeByValue(int n) {
   return n * n * n; // cube local variable n and return result
}
```

```
The original value of number is 5
The new value of number is 125
```

Pass by reference: pointers enable pass by reference.

When calling a function with arguments that should be modified in the caller, you use & to pass each variable's address.

Arrays are not passed using operant & because an array name is equivalent to &arrayName[0].

A function that receives the address of a variable in the caller can use the indirection operator (*) to modify the value at that location in the caller's memory, thus effecting pass-by-reference.

```c
// fig07_03.c
// Cube a variable using pass-by-reference with a pointer argument.

#include <stdio.h>

void cubeByReference(int *nPtr); // function prototype

int main(void) {
   int number = 5; // initialize number

   printf("The original value of number is %d", number);
   cubeByReference(&number); // pass address of number to cubeByReference
   printf("\nThe new value of number is %d\n", number);
}

// calculate cube of *nPtr; actually modifies number in main
void cubeByReference(int *nPtr) {
   *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}
```

```
The original value of number is 5
The new value of number is 125
```

Step 2: After cubeByValue receives the call:

```
int main(void) {                    number
                                      5
   int number = 5;
   number = cubeByValue(number);
}

int cubeByValue( int n ) {
   return n * n * n;                    n
}                                       5
```

Step 3: After cubeByValue cubes parameter n and before cubeByValue returns to main:

```
int main(void) {                    number
                                      5
   int number = 5;
   number = cubeByValue(number);
}

int cubeByValue(int n) {
                  125
   return n * n * n;                    n
}                                       5
```

Fig. 7.4 | Analysis of a typical pass-by-value. (Part 1 of 2.)

Step 4: After cubeByValue returns to main and before assigning the result to number:

```
int main(void) {                    number
                                      5
   int number = 5;
              125
   number = cubeByValue(number);
}
```

Step 5: After main completes the assignment to number:

```
int main(void) {                    number
                                     125
   int number = 5;
      125          125
   number = cubeByValue(number);
}
```

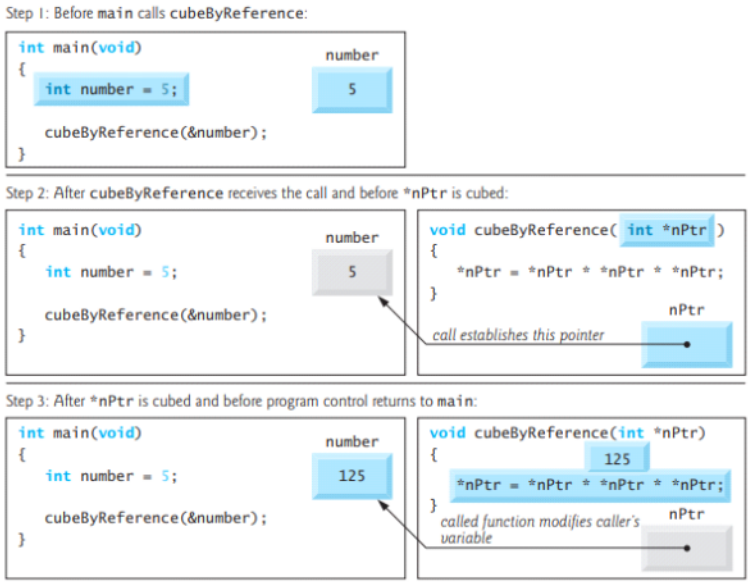
//

Pass-By-Reference

Line 12 of Fig. 7.3 passes the variable number's address to function cubeByReference (lines 17–19)—passing the address enables pass-by-reference. The function's parameter is a pointer to an int called nPtr (line 17). The function uses the expression *nPtr to dereference the pointer and cube the value to which it points (line 18). It assigns the result to *nPtr—which is really the variable number in main—thus changing number's value in main. Use pass-by-value unless the caller explicitly requires the called function to modify the argument variable's value in the caller. This prevents accidental modification of the caller's arguments and is another example of the principle of least privilege.

A function receiving an adress as an argument must recive it in a pointer parameter.
The function prototype for cubeByReference (line 6) specifies an int * parameter. As with other parameters, it's not necessary to include pointer names in function prototypes—they're ignored by the compiler—but it's good practice to include them for documentation purposes.

For a function that expects a one-dimensional array argument, the function's prototype and header can use the pointer notation shown in the parameter list of function cubeByReference (line 17). The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array. So, the function must "know" when it's receiving an array vs. a single variable passed by reference. When the compiler encounters a function parameter for a one-dimensional array of the form int b[], the compiler converts the parameter to the pointer notation int *b. The two forms are interchangeable. Similarly, for a parameter of the form const int b[] the compiler converts the parameter to const int *b.



USING THE CONST QUALIFIER

There are four ways to pass to a function a pointer to data:
• a non-constant pointer to non-constant data.
The highest level of data access is granted by a non-constant pointer to non-constant data. The data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items. A function might use such a pointer to receive a string argument, then process (and possibly modify) each character in the string.

```c
// fig07_06.c
// Converting a string to uppercase using a
// non-constant pointer to non-constant data.
#include <ctype.h>
#include <stdio.h>

void convertToUppercase(char *sPtr); // prototype

int main(void) {
   char string[] = "cHaRaCters and $32.98"; // initialize char array

   printf("The string before conversion is: %s\n", string);
   convertToUppercase(string);
   printf("The string after conversion is: %s\n", string);
}

// convert string to uppercase letters
void convertToUppercase(char *sPtr) {
   while (*sPtr != ) { // current character is not
      *sPtr = toupper(*sPtr); // convert to uppercase
      ++sPtr; // make sPtr point to the next character
   }
}
```

```
The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

• a non-constant pointer to constant data.
A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified.
A function might receive such a pointer to process an array argument elements without modifying them.
For ex.
Printcharacters, declares parameter sprt to be of type const char
. The declaration is read from right to left as “sPtr is a pointer to a character constant.” The function’s for statement outputs each character until it encounters a null character. After displaying each character, the loop increments pointer sPtr to point to the string’s next character.

```c
// fig07_07.c
// Printing a string one character at a time using
// a non-constant pointer to constant data.

#include <stdio.h>

void printCharacters(const char *sPtr);

int main(void) {
   // initialize char array
   char string[] = "print characters of a string";

   puts("The string is:");
   printCharacters(string);
   puts("");
}

// sPtr cannot be used to modify the character to which it points,
// i.e., sPtr is a "read-only" pointer
void printCharacters(const char *sPtr) {
   // loop through entire string
   for (; *sPtr != ; ++sPtr) { // no initialization
      printf("%c", *sPtr);
   }
}
```

```
The string is:
print characters of a string
```

• a constant pointer to non-constant data.

A constant pointer to non-constant data always points to the same memory location, but the data at that location can be modified through the pointer. This is the default for an array name, which is a constant pointer to the array's first element. All data in the array can be accessed and changed by using the array name and array subscripting.
Can be used to receive an array as argument to a function.
Pointers that are declared const must be initialized when they're defined.

```c
// fig07_09.c
// Attempting to modify a constant pointer to non-constant data.
#include <stdio.h>

int main(void) {
   int x = 0; // define x
   int y = 0; // define y

   // ptr is a constant pointer to an integer that can be modified
   // through ptr, but ptr always points to the same memory location
   int * const ptr = &x;

   *ptr = 7; // allowed: *ptr is not const
   ptr = &y; // error: ptr is const; cannot assign new address
}
```

Microsoft Visual C++ Error Message

```
fig07_09.c(14,4): error C2166: l-value specifies const object
```

• a constant pointer to constant data.
The least access privilege is granted by a constant pointer to constant data. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified. This is how an array should be passed to a function that only looks at the array's elements using array subscript notation and does not modify the elements.

```c

int main(void) {
   int x = 5;
   int y = 0;

   // ptr is a constant pointer to a constant integer. ptr always
   // points to the same location; the integer at that location
   // cannot be modified
   const int *const ptr = &x; // initialization is OK

   printf("%d\n", *ptr);
   *ptr = 7; // error: *ptr is const; cannot assign new value
   ptr = &y; // error: ptr is const; cannot assign new address
}
```

Microsoft Visual C++ Error Message

```
fig07_10.c(15,5): error C2166: l-value specifies const object
fig07_10.c(16,4): error C2166: l-value specifies const object
```

SIZEOF()

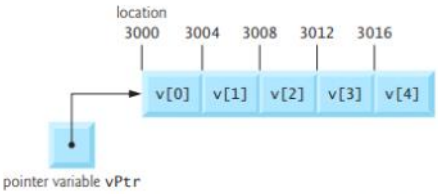
Determines size of a datatype in bytes.
This operator is applied at compilation time unless its operand is a variable-length array (VLA)
Number of element in array can also be determined with sizeof:

Double real[22];
Sizeof(real) / sizeof(real[0])
The expression divides the array real's number of bytes by the number of bytes used to store one element of the array (a double value). This calculation works only when using the actual array's name, not when using a pointer to the array

POINTER OPERATIONS

Allowed:
++ or --
Adding an integer to pointer
Subtracting an int from a pointer
Subtracting a pointer from another (only when both pointer point into the same array)

Assume the array `int v[5]` is defined, and its first element is at location 3000 in memory. Also, assume the pointer `vPtr` points to `v[0]`—so the value of `vPtr` is 3000. The following diagram illustrates this scenario for a machine with four-byte integers:

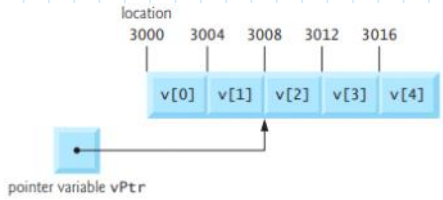


The variable `vPtr` can be initialized to point to array `v` with either of the statements

```
vPtr = v;
vPtr = &v[0];
```

In conventional arithmetic, 3000 + 2 yields the value 3002. This is normally not the case with pointer arithmetic. When you add an integer to or subtract one from a pointer, the pointer increments or decrements by that integer times the size of the object to which the pointer refers.

vPtr += 2;
would produce 3008 (3000 + 2 * 4), assuming an integer is stored in four bytes of memory. In the array v, vPtr would now point to v[2], as in the following diagram:



If vPtr had been incremented to 3016 (v[4]), the statement
vPtr -= 4;
would set vPtr back to 3000 (v[0])—the beginning of the array.
Using pointer arithmetic to adjust pointers to point outside an array's bounds is a logic error that could lead to security problems.

++vptr; increment the pointer to the next array element
Vptr++ increment the pointer to the next array element
--vptr; decrement the pointer to point to the previous element
Vptr--; decrement the pointer to point to the previous element

You can compare pointers using equality and relational operators, but such comparisons are meaningful only if the pointers point to elements of the same array; otherwise, such comparisons are logic errors

If `vPtr` contains the location 3000 and `v2Ptr` contains the address 3008, the statement

```
x = v2Ptr - vPtr;
```

assigns to `x` the *number of array elements* between `vPtr` and `v2Ptr`, in this case, 2 (not 8). Pointer arithmetic is undefined unless performed on elements of the same array. We cannot assume that two variables of the same type are stored side-by-side in memory unless they're adjacent elements of an array.

Arrays and pointers are intimately related and often may be used interchangeably.
You can think of an array name as a constant pointer to the array's first element.
Pointers can be used to do any operation involving array subscripting.
Assume the following definitions:
int b[5]; int *bPtr;
Because the array name b (without a subscript) is a pointer to the array's first element, we can set bPtr to the address of the array b's first element with the statement:
bPtr = b; This is equivalent to taking the address of array b's first element as follows:
bPtr = &b[0];

Array element b[3] can alternatively be referenced with the pointer expression
*(bPtr + 3)
The 3 in the expression is the offset to the pointer. When bPtr points to the array's first element, the offset indicates which array element to reference—the offset's value is identical to the array subscript
7.9,7.10

FUNCTION POINTER

A function name is really the starting adress in memory of the code that performs the functions taks.
A pointer to a function contains the adress of the function in memory.
Pointers to functions can be passed to functions. Returned to functions, stored in arrays, assigned to other function pointers of the same type and compared with one another for equality or inequality.

Type (p*functionname)(type of paramenters);
We can assign the value of a function pointer as follow

Pfunctionname = functionname;

```c
int sum(int, int);
int product(int, int);
int difference(int, int);

int main(void)
{
    int a = 10;                          // Initial value for a
    int b = 20;                           // Initial value for b
    int result = 0;                      // Storage for results
    int (*pfunction)(int, int);              // Function pointer declaration

    pfunction = sum;                         // Points to function sum()
    result = pfunction(a, b);                // Call sum() through pointer
    printf("pfunction = sum result = %2d\n", result);

    pfunction = product;                     // Points to function product()
    result = pfunction(a, b);                // Call product() through pointer
    printf("pfunction = product result = %2d\n", result);

    pfunction = difference;                  // Points to function difference()
    result = pfunction(a, b);                // Call difference() through pointer
    printf("pfunction = difference result = %2d\n", result);

    return 0;
}
```