C under Linux

Dr. Naeem Odat

Department of Computer and Communications Engineering

C - Pointers

C - Pointers

C - Pointers

Pointer

Variable whose value is the **address** of another variable, i.e., **direct address** of the memory location.

Syntax

```
type *var-name;
```

Examples

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

C - Pointers

Pointer Variable Declarations and Initialization

```
#include<stdio.h>
int main(){
    int i;
    int j=20;
    int *p=&i;
    i=200;
    printf("i is: %d\n",i);
    printf("Address of i is: %p\n",&i);
    printf("p has: %p\n", p);
    printf("Address of p is: %p\n",&p);
    printf("*p has: %d\n",*p);
    return 0;
}
```

- Can declare pointers to any data type. Takes fixed number of bytes space.
- Initialize pointers to 0, NULL, or an address (NULL preferred).

C - Pointers

What is the output?

```
#include <stdio.h>
int main(){
    int i = 10;
    int j = 20;
    int* p = &i;
    printf("*p = %d\n", *p);
    *p = 200;
    printf("Value of i is = %d\n", i);
    return(0);
}
```

C - Pointers

What is the output?

```
#include <stdio.h>
int main(){
    int i=200;
    int* p;
    *p=250;//error
    return(0);
}
```

```
#include <stdio.h>
int main(){
    int i=200;
    int* p;
    p=&i;
    *p=250;
    return(0);
}
```

C - Pointers

What is the output?

```
char *pstr = NULL, str[20] = "Hello World!";
int *pnums = NULL, nums[10] = {1,2,3,4,5,6,7,8,9,0};
double *preals = NULL, reals[10] = {1.2,3.4,4.3,5.0,0.4};
pstr = str;
pnums = nums;
preals = reals;
pnums++;//What is it pointing to?
```

- At this point:

```
pstr = &str[0] and *pstr = ’H’
pnums= &nums[0] and *pnums = 1
preals = &reals[0] and *preals = 1.2
```

C - Pointers

Pointer Arithmetic

- For the program:

```
#include<stdio.h>
int main(){
    double *realsP, reals[8] = {1.2, 3.4, 4.3, 5.0, 0.4};
    for (realsP = reals; realsP <= &reals[7]; realsP++)
        printf("address: %p \t value %g \n", realsP, *realsP
    return(0);
}
```

- The output will be something like:

```
address: 65fda4     value 1.2
address: 65fdac     value 3.4
address: 65fdb4     value 4.3
address: 65fdbc     value 5
address: 65fdc4     value 0.4
address: 65fdcc     value 0
address: 65fdd4     value 0
address: 65fddc     value 0
```

C - Pointers

Pointer Arithmetic

```
int nArr[5]={1, 2, 3, 4, 5};
int *pArr = nArr;
while(pArr <= &nArr[4]){
    printf("%d ", *pArr);
    pArr++;
}
```

C - Pointers

Pointer Arithmetic

- Subtracting pointers:
  - Returns the number of elements between the pointers.
  - If v is an array of 10 elements, vPtr2 $=$ &v[2] and vPtr $=$ &v[0], then (vPtr2 - vPtr) would produce 2.
- Pointer comparison $(<, ==, >)$:
  - See which pointer points to the higher numbered array element.
  - Also, see if a pointer points to 0 or NULL.

C - Pointers

Pointer Arithmetic

- Pointers of the same type can be assigned to each other.
- If not the same type, a cast operator must be used.
- Exception: pointer to void (type void*).
- Generic pointer (void*), represents any type.
- No casting needed to convert a pointer to void pointer.
- void pointers cannot be used to refer to the memory location which it points to (cast first).

C - Pointers

Pointers and Arrays

- Arrays and pointers are closely related
- Array name is like a constant pointer.
- Pointers can do array sub-scripting operations.

  ```
  int b[5]={1,2,3,4,5};
  int *bPtr=b;//set them equal to one another
  ```
- The array name (b) is actually the address of the array first element.
- bPtr = &b[0] is another way to point to the array. Explicitly assigns bPtr to address of first element of b.

C - Pointers

Pointers and Arrays

Element b[n] can be accessed by:

- *(bPtr + n), where n is the offset. Called **pointer/offset notation**.
- bPtr[n], bPtr[n] same as *(bPtr +n) or b[n].
- Performing pointer arithmetic on the array itself *(b + n).

C - Pointers

What is wrong?

```
#include<stdio.h>
int main(){
    int* i, j;
    *j = 100;
    return(0);
}
```

C - Pointers

What is the output?

```
#include<stdio.h>
void abc(int*);
int main(){
    int a[5];
    abc(a);
    printf("%d\n", a[1]);
    return(0);
}

void abc(int* a){
    a++;
    *a = 100;
}
```

C - Pointers

const with pointer

```
#include<stdio.h>
int main(void){
    int nArr[] = {1, 2, 3, 4, 5};
    const int* pA = nArr;
    int* const pB = nArr;
    const int* const pC = nArr;
    ...
    return(0);
}
```

- The value to which pA points to is made constant, so we cannot change the value of *pA. We can change the value of pA.
- pB is made constant, hence pB cannot denote other elements in the array, except for the first one. The value of *pB can be changed.
- In the case of pC, both the pC and *pC are constant.

C - Pointers

Pointers and multidimensional arrays

- c stores two-dimensional arrays in row-major order; in other words, the elements of row 0 come first, followed by the elements of row 1, and so forth.
- We can take advantage of this layout when working with pointers.
- Make a pointer p points to the first element in a two-dimensional array (the element in row 0, column 0).
- We can visit every element in the array by incrementing p repeatedly.

C - Pointers

Pointers and multidimensional arrays

- To initialize all elements of a two-dimensional array to zero:

```
int arr[NUM_ROWS][NUM_COLS];
int row, col;

for (row = 0; row < N_ROWS; row++)
    for (col = 0; col < N_COLS; col++)
        arr[row][col] = 0;
```

- Using pointer:

```
int *p;
for(p=&arr[0][0];p<=&arr[NUM_ROWS-1][NUM_COLS-1];p++)
    *p = 0;
```

C - Pointers

What is the output?

```
#include<stdio.h>
int main(){
    char cStr[3][3] = {{’a’,’b’,’c’},"def","gh"};
    printf("%s\n", cStr);
    return 0;
}//abcdefgh
```

C - Pointers

void pointer

```
int x = 10;
int* pn1 = &x;
void* pv1;
int* pn2;
pv1 = pn1;
*pv1= 11;//Error derefrencing pv1 is not allowed
*pn1 = 12;
pn2 = pv1;
*pn2 = 13;
```

C - Pointers

Use of a **void** pointer

As a function argument:

```
memcpy(void* s1, const void* s2, size_t n)
```

- Copies n characters from object pointed to by s2 into object pointed to by s1.
- size_t specifies the number of bytes the function will process.

C - Pointers

Use of a **void** pointer

As a return value from a function:

```
int* pX = (...) malloc(...);
```

- Returns a pointer to a block of memory.
- Return type should be generic

C - Pointers

Use of a **void** pointer

```
void*CreateCar(int nCh){
    void* pV = NULL;
    switch(nCh){
        case 1:
            pV=malloc(sizeof(HONDA)*1);
            break;
        case 2:
            pV=malloc(sizeof(BMW)*1);
            break;
    }
    return pV;
}
NOTE: BMW and HONDA are user defined structures.
//When used the caller to type cast appropriately
HONDA* pM = (HONDA*)CreateCar(1);
BMW* pA = (BMW*)CreateCar(2);
```

C - Pointers

What is the output?

```
#include<stdio.h>
int main(void){
    int i = 100;
    void* p = &i;

    printf("\n%d\n", *p);
    return(0);
}
```

C - Pointers

Notice

```
char* foo(void){
    static char ca[10];
    return ca;
}
```

- Anyone calling this function has access to this block. Could be dangerous.