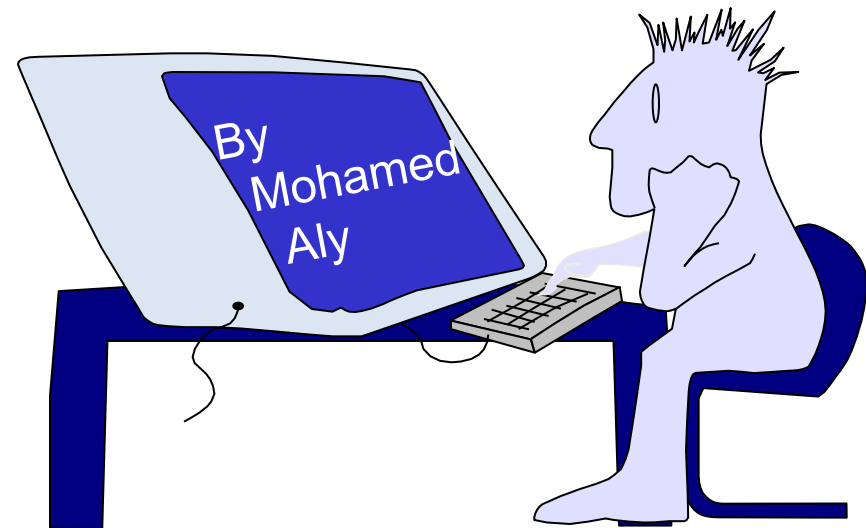


Embedded C

C Programming Part 4



Pointers operators

- Arithmetic operations can be performed on pointers
 - Increment/decrement pointer (++ or --)
 - Add an integer to a pointer(+ or += , - or -=)
 - Pointers may be subtracted from each other
 - Operations meaningless unless performed on an array
 - More about this later
- Pointer comparison (< , <= , == , != , > , >=)
- 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, represents any type
 - No casting needed to convert a pointer to void pointer
 - void pointers can not be dereferenced
 - Can't use pointer arithmetic

Pointers operators(Cont.)

- Since a pointer is just a memory address, we can add to it to traverse an array
- (p+1) returns a pointer to the next array element
- `x = *++p; /* p = p + 1 ; x = *p ; */`
- (*p++) versus ((*p)++)
 - `x = *p++; /* x = *p ; p = p + 1; */`
`/* (*p++) ? *(p)++ ? *(p++) */`
 - `x = (*p)++; /* x = *p ; *p = *p + 1; */`
- In reality, p+1 doesn't add 1 to the memory address, it adds the size of the array element

```
int get(int array[], int n)
{
    return (array[n]); /* This is equivalent to return *(array + n); */
}
```

Pointers operators(Cont.)

How many of the following are invalid?

pointer + integer

integer + pointer

pointer + pointer

pointer - integer

integer - pointer

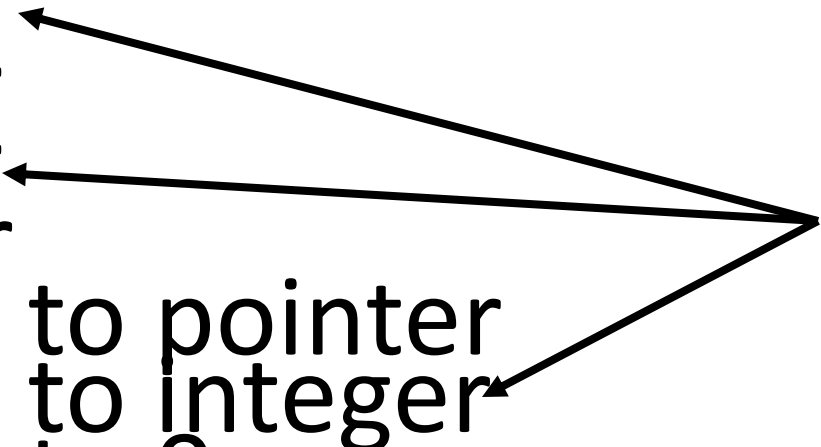
pointer - pointer

compare pointer to pointer

compare pointer to integer

compare pointer to 0

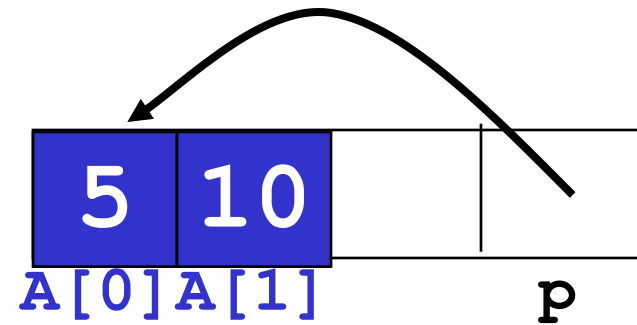
compare pointer to **NULL**



Pointers operators(Cont.)

Question

```
int main( void)
{
    int A[] = {5,10};
    int *p = A;
    printf("%u %d %d %d\n",p,*p,A[0],A[1]);
    p = p + 1;
    printf("%u %d %d %d\n",p,*p,A[0],A[1]);
    * p = *p + 1;
    printf("%u %d %d %d\n",p,*p,A[0],A[1]);
}
```



- If the first printf outputs **100 5 5 10**, what will the other two printf outputs?
 - a) 101 10 5 10 then 101 11 5 11
 - b) 104 10 5 10 then 104 11 5 11
 - c) 101 <other> 5 10 then 101 <3-others>
 - d) 104 <other> 5 10 then 104 <3-others>
 - e) One of the two printfs causes an ERROR

Pointers operators(Cont.)

Example: Copying arrays

```
void copy (int *from, int *to, int n)
{
    int i;
    for(i=0; i<n; i++)
    {
        *to++ = *from++;
    }
}
```

1st time looping

2nd time looping

3rd time looping

Assume N=3;

from	to
10	?
20	?
30	?

Pointers operators(Cont.)

Example

- Sometimes we want to have a procedure increment a variable?

```
void AddOne(int x)
{
    x = x + 1;
}
int main(void)
{
    int y = 5;
    AddOne( y);
    printf("y = %d\n", y);
    return 0;
}
```

**"Y=5" gets
printed**

Why!!!

**Because x takes a
copy of y but it has
not any access to y
itself**

Pointers operators(Cont.)

Example(Cont.)

```
void AddOne(int * x)
{
    *x = *x + 1;
}

int main(void)
{
    int y = 5;
    AddOne( &y);
    printf("y = %d\n", y);
    return 0;
}
```

**"Y=6" gets
printed**

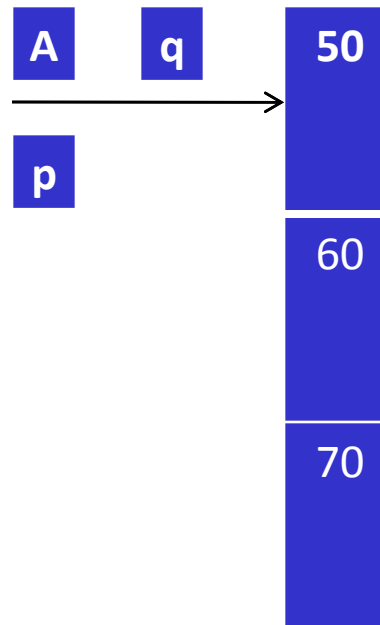
Why!!!

**Because x takes a
copy of y address so
it has access to y
itself**

Pointers operators(Cont.)

Example(Cont.)

```
void IncrementPtr (int *p)
{
    p = p + 1;
}
int main(void)
{
    int A[3] = {50, 60, 70};
    int *q = A;
    IncrementPtr(q);
    printf("*q = %d\n", *q);
}
```



**"*q = 50"
gets
printed**

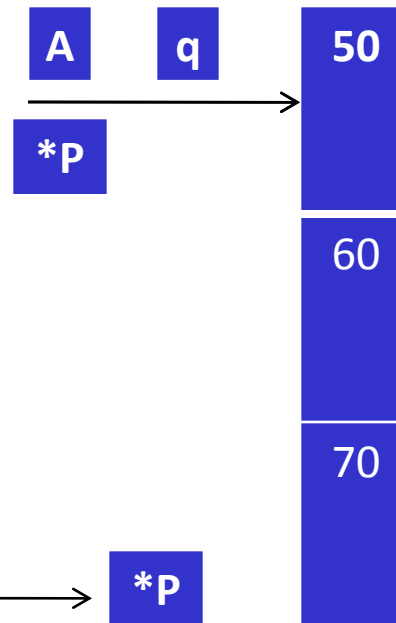
Why!!!

**Because p takes a
copy of q and p has
access to content of
address referenced
by q but it has not
any access to q itself**

Pointers operators(Cont.)

Example(Cont.)

```
void IncrementPtr (int **p)
{
    *p = *p + 1;
}
int main(void)
{
    int A[3] = {50, 60, 70};
    int *q = A;
    IncrementPtr(&q);
    printf("*q = %d\n", *q);
}
```



***q = 60**

Why!!!

Because p takes a
copy of q address so
it has access to q
itself

Pointers operators(Cont.)

- So what's valid pointer operations?
 - Add an integer to a pointer.
 - Subtract 2 pointers (in the same array).
 - Compare pointers (<, <=, ==, !=, >, >=)
 - Compare pointer to **NULL** (indicates that the pointer points to nothing).
- Everything else is illegal since it makes no sense:
 - adding two pointers
 - multiplying pointers
 - subtract pointer from integer

Pointers pros and cons

- Why use pointers?
 - If we want to pass a huge array, it's easier and faster to pass a pointer than the whole thing
 - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
 - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them
 - **Wilding reference**: wild pointer which are not initialized during its definition holding some junk value(a valid address) are Wild pointer.
 - **Dangling reference** (premature free) A dangling pointer is a (Non NULL) pointer that points to invalid data or to data which is not valid anymore
 - **Memory leaks** (tardy free) a memory leak happen when an object is stored in memory but cannot be accessed by the running code

Pointers and allocation

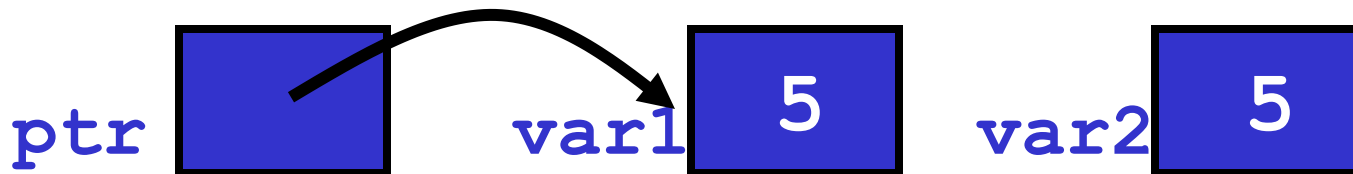
- After declaring a pointer, pointer doesn't actually point to anything yet (*it actually points somewhere - but don't know where!*)
- We can either:
 - make it point to something that already exists
 - allocate room in memory for something new that it will point to...

Pointers and allocation(Cont.)

- Pointing to something that already exists:

```
int *ptr, var1, var2;  
var1 = 5;  
ptr = &var1;  
var2 = *ptr;
```

- var1** and **var2** have room implicitly allocated for them.



sizeof operator

- C has operator **sizeof()** which gives size in bytes of type or variable
- Assume size of objects can be misleading and is bad style, so use **sizeof(type)**
- **sizeof** knows the size of arrays
- As well knows for arrays whose size is determined at run-time

Dynamic memory allocation

- To allocate room for something new to point to, use **malloc()**
- **malloc** takes number of bytes to allocate(**sizeof**) and returns pointer of type void *. If no memory available, it returns **NULL**

```
ptr = (int *) malloc (sizeof(int));
```

- Now, ptr points to a space somewhere in memory of size (**sizeof(int)**) in bytes

- **malloc** is almost never used for 1 variable

```
ptr = (int *) malloc (n*sizeof(int));
```

- This allocates an array of n integers

Dynamic memory allocation(Cont.)

- Once **malloc()** is called, the memory location contains garbage, so do not use it until you have set its value
- After dynamically allocating space, we must dynamically free it
- **free** deallocates memory allocated by **malloc**
- **free** takes a pointer to memory that will be deallocated as an argument

free(ptr);

- The program frees all memory on exit (or when main returns), don't be lazy
 - You never know when your main will get transformed into a subroutine

Dynamic memory allocation(Cont.)

- The following two things will cause your program to crash or behave strangely later on, and cause **very hard** to figure out bugs
 - freeing the same piece of memory twice
 - calling **free()** on something you didn't get back from **malloc()**
- The runtime does not check for these mistakes
 - The Memory allocation is so performance-critical that there is not time to do this
 - The usual result is that you corrupt the memory allocator's internal structure
 - You won't find out until much later on, in a totally unrelated part of your code!!

Dynamic memory allocation(Cont.)

```
calloc( nmembers, size );
```

- *nmembers* – number of elements
- *size* – size of each element
- Returns a ***pointer to a dynamic array, each of size***
- initializes contents of memory to zeroes
- Zeroing out the memory may take a little time, so you probably want to use **malloc()** if that performance is an issue
- Allocated memory may/may not be contiguous

Dynamic memory allocation(Cont.)

realloc(*pointerToObject*, *newSize*)

- *pointerToObject* – pointer to the object being reallocated
- *newSize* – new size of the object
- Returns pointer to reallocated memory
- Returns **NULL** if cannot allocate space
- If *newSize* equals 0 then the object pointed to is freed
- If *pointerToObject* equals 0 then it acts like **malloc**

How to overcome dangling pointers and wild pointers?

- Initialize pointer variable to NULL
- After deallocating memories initialize pointer to NULL
- Take care or out of scope pointers (Block/Function)

String

- A string in C is just an array of characters
`char string[] = "abc";`
- How do you tell how long a string is?
 - Last character is followed by a **0** byte (**null terminator**)
- One common mistake is to forget to allocate an extra byte for the null terminator
- More generally, C requires the programmer to manage memory manually
 - When creating a long string by concatenating several smaller strings, the programmer must insure there is enough space to store the full string
 - What if you don't know ahead of time how big your string will be?
 - Buffer overrun security holes

String(Cont.)

Standard functions

- **int strlen(char *string);**
 - computes the length of string
- **int strcmp(char *str1, char *str2);**
 - returns 0 if str1 and str2 are identical (how is this different from str1 == str2?)
- **char *strcpy(char *dst, char *src);**
 - copies the contents of string src to the memory at dst. The caller must ensure that dst has enough memory to hold the data to be copied.

Thanks

Embedded C