

Pointers

Ritankar Mandal

Pointer

A pointer is a variable that contains the address of a variable.

```
1 int x = 1, y = 2, z[10];
2 int *ip; /* ip is a pointer to int */
3
4 ip = &x; /* ip now points to x */
5 y = *ip; /* y is now 1 */
6 *ip = 0; /* x is now 0 */
7 ip = &z[0]; /* ip now points to z[0] */
```

The `&` operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

Pointer

Operators `*` and `&` bind more tightly than arithmetic operators, so the following command takes whatever `ip` points at, adds 1, and assigns the result to `y`.

```
y = *ip + 1
```

Each command below increments what `ip` points to.

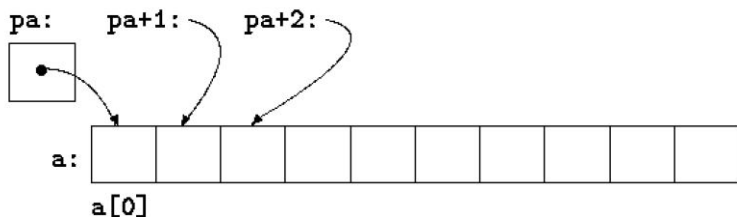
```
*ip += 1
```

```
++*ip
```

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment `ip` instead of what it points to, because unary operators like `*` and `++` associate right to left.

Array



```
int a[10];
```

defines an array of size 10, – a block of 10 consecutive objects named `a[0]`, `a[1]`, ..., `a[9]`. `a[i]` refers to the i -th element of the array. If `pa` is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets `pa` to contain the address of `a[0]`.

Pointers and Function Arguments

```
1 void swap(int x, int y) /* WRONG */
2 {
3     int temp;
4     temp = x;
5     x = y;
6     y = temp;
7 }
```

Pointers and Function Arguments

```
1 void swap(int x, int y) /* WRONG */
2 {
3     int temp;
4     temp = x;
5     x = y;
6     y = temp;
7 }
```

C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. `swap` can't affect the arguments *a* and *b* in the routine that called it. The function above swaps **copies** of *a* and *b*.

Pointers and Function Arguments

```
1 void swap(int *px, int *py) /* interchange *px and *py */
2 {
3     int temp;
4     temp = *px;
5     *px = *py;
6     *py = temp;
7 }
```

The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

```
swap(&a, &b);
```

Pointers and Function Arguments

```
1 void increaseI(int n)
2 {
3     n++;
4     printf("value (inside function) = %d \n", n);
5 }
6
7 int main()
8 {
9     int i = 1;
10    printf("value = %d \n", i);
11
12    increaseI(i);
13    printf("value = %d \n", i);
14    return 0;
15 }
```


Pointers and Function Arguments

```
1 void increaseIbyAddress(int *nAddress)
2 {
3     (*nAddress)++;
4     printf("value (inside function) = %d \n",
5           (*nAddress));
6 }
7 int main()
8 {
9     int i = 1;
10    printf("value = %d \n", i);
11
12    increaseIbyAddress(&i);
13    printf("value = %d \n", i);
14    return 0;
15 }
```

Pointers and Arrays

`a[i]` can also be written as `*(a+i)`. C converts `a[i]` to `*(a+i)`.

As formal parameters in a function definition, the two are equivalent.

```
int s[];  
int *s;
```

Command-line Arguments

This is a way to pass command-line arguments to a program when it begins executing. When `main` is called, it is called with two arguments.

The first, called `argc`, for argument count, is the number of command-line arguments the program was invoked with.

The second, `argv`, for argument vector, is a pointer to an array of character strings that contain the arguments, one per string.

```
1 #include<stdio.h>
2 int main(int argc, char *argv[])
3 {
4     int i;
5     for (i = 1; i < argc; i++)
6         printf("%s, ", argv[i]);
7     return 0;
8 }
```

Address Arithmetic

If p is a pointer to some element of an array, then $p++$ increments p to point to the next element, and $p+=i$ increments it to point i elements beyond where it currently does.

Address Arithmetic

If p is a pointer to some element of an array, then $p++$ increments p to point to the next element, and $p+=i$ increments it to point i elements beyond where it currently does.

If p and q point to members of the same array, then relations like $==$, $!=$, $<$, $<=$, $>$, $>=$, etc., work properly. But the behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array.

Address Arithmetic

A pointer and an integer may be added or subtracted.

$p + n$

means the address of the n -th object beyond the one p currently points to. This is true regardless of the kind of object p points to; n is scaled according to the size of the objects p points to, which is determined by the declaration of p . If an `int` is four bytes, for example, the `int` will be scaled by four.

Address Arithmetic

A pointer and an integer may be added or subtracted.

$p + n$

means the address of the n -th object beyond the one p currently points to. This is true regardless of the kind of object p points to; n is scaled according to the size of the objects p points to, which is determined by the declaration of p . If an `int` is four bytes, for example, the `int` will be scaled by four.

Pointer subtraction is also valid: if p and q point to elements of the same array, and $p < q$, then $(q - p + 1)$ is the number of elements from p to q inclusive.

Address Arithmetic

A pointer and an integer may be added or subtracted.

$p + n$

means the address of the n -th object beyond the one p currently points to. This is true regardless of the kind of object p points to; n is scaled according to the size of the objects p points to, which is determined by the declaration of p . If an `int` is four bytes, for example, the `int` will be scaled by four.

Pointer subtraction is also valid: if p and q point to elements of the same array, and $p < q$, then $(q - p + 1)$ is the number of elements from p to q inclusive.

All other pointer arithmetic is illegal. It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add `float` or `double` to them.

Character Pointers and Functions

```
char amessage[] = "now is the time"; /* an array */  
char *pmessage = "now is the time"; /* a pointer */
```

`amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage. On the other hand, `pmessage` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.

Character Pointers and Functions

```
1  /* strcpy: copy t to s; array subscript version */
2  void strcpy(char *s, char *t)
3  {
4      int i;
5      i = 0;
6      while((s[i] = t[i]) != '\0')
7          i++;
8  }
9  /* strcpy: copy t to s; pointer version */
10 void strcpy(char *s, char *t)
11 {
12     int i;
13     i = 0;
14     while((*s = *t) != '\0')
15     {
16         s++;
17         t++;
18     }
19 }
```

Character Pointers and Functions

```
1  /* strcpy: copy t to s; pointer version 2 */
2  void strcpy(char *s, char *t)
3  {
4      while ((*s++ = *t++) != '\0')
5          ;
6  }
7  /* strcpy: copy t to s; pointer version 3 */
8  void strcpy(char *s, char *t)
9  {
10     while (*s++ = *t++)
11         ;
12 }
```

The value of `*t++` is the character that `t` pointed to before `t` was incremented; the postfix `++` doesn't change `t` until after this character has been fetched. The net effect is that characters are copied from `t` to `s`, up and including the terminating `'\0'`.

Complicated Declarations

The difference between the following two illustrates the problem.

```
int *f();  
int (*pf)();
```

`f` is a function returning pointer to `int`. `pf` is a pointer to function returning `int`. `*` is a prefix operator and it has lower precedence than `()`, so parentheses are necessary to force the proper association.

Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on.

```
1  /* A normal function with an int parameter and void
   2     return type */
   3  void fun(int a)
   4  {
   5     printf("Value of a is %d\n", a);
   6  }
```

Pointers to Functions

```
1  int main()
2  {
3      int i;
4      printf("Enter an integer: ");
5      scanf("%d", &i);
6
7      /* fun_ptr is a pointer to function fun() */
8      void (*fun_ptr)(int) = &fun;
9
10     /* The above line is equivalent of following two
11     void (*fun_ptr)(int);
12     fun_ptr = &fun; */
13
14     fun(i);
15     /* Invoking fun() using fun_ptr */
16     (*fun_ptr)(i);
17
18     return 0;
19 }
```