Pointers

Ritankar Mandal

Pointer

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

```c
int x = 1, y = 2, z[10];
int *ip; /* ip is a pointer to int */

ip = &x; /* ip now points to x */
y = *ip; /* y is now 1 */
*ip = 0; /* x is now 0 */
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.

Pointers and Function Arguments

```c
void swap(int x, int y) /* WRONG */
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Pointers and Function Arguments

```c
void swap(int x, int y) /* WRONG */
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

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

```c
void swap(int *px, int *py) /* interchange *px and *py */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

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

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

Pointers and Function Arguments

```c
void inceaseI(int n)
{
    n++;
    printf("value (inside function) = %d \n", n);
}

int main()
{
    int i = 1;
    printf("value = %d \n", i);

    inceaseI(i);
    printf("value = %d \n", i);
    return 0;
}
```

Pointers and Function Arguments

```c
void inceaseIbyAddress(int *nAddress)
{
    (*nAddress)++;
    printf("value (inside function) = %d \n",
        (*nAddress));
}

int main()
{
    int i = 1;
    printf("value = %d \n", i);

    inceaseIbyAddress(&i);
    printf("value = %d \n", i);
    return 0;
}
```

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.

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

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

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while((s[i] = t[i]) != ’\0’)
        i++;
}
/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while((*s = *t) != ’\0’)
    {
        s++;
        t++;
    }
}
```

Character Pointers and Functions

```c
/* strcpy: copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != ’\0’)
        ;
}
/* strcpy: copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

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.

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

Pointers to Functions

```c
int main()
{
    int i;
    printf("Enter an integer: ");
    scanf("%d", &i);

    /* fun_ptr is a pointer to function fun() */
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
    void (*fun_ptr)(int);
    fun_ptr = &fun; */

    fun(i);
    /* Invoking fun() using fun_ptr */
    (*fun_ptr)(i);

    return 0;
}
```