C Lecture 3

Pointers in C

C has a very powerful feature, and if used incorrectly a very dangerous feature, which allows a program (at run-time) to access the addresses of its memory. This is supported in the language through the use of pointers. There is much written about the power and expressiveness of C’s pointers, and much (more recently) written about Java’s lack of pointers. More precisely, Java has pointers, termed references, but the references to Java’s objects are consistently and carefully constrained at both compile and run-time, so very little can go wrong.

C has both “standard” variables and structures, and pointers to these variables and structures (Java has only references to objects, and it is possible to manipulate objects only by using references). C’s drawback is that while the pointers allow us to easily refer to scalar variables and aggregate structures, C has very little support to prevent us accessing anything else (accidentally or otherwise) at run-time. The speed advantage provided by the availability of pointers is trivially consumed by the time taken to debug a program that uses pointers incorrectly.

C’s pointers allow us to refer to the address of a variable rather than its value. If this were all that were possible, we may be able to get away without using pointers at all. “Unfortunately”, parameters to C’s functions may only be passed by value, and so a rudimentary understanding of C’s pointers is needed to use “pass-by-reference” parameter passing in C.

Consider the following example, trying to interchange the values of two integer variables:

```c
#include <stdio.h>

void swap(int i, int j)
{
    int temp;
    temp = i;
    i = j;
    j = temp;
}
```
int main()
{
    int a=3, b=5;

    printf("before a=%d, b=%d\n",a,b);
    swap(a,b);
    printf("after a=%d, b=%d\n",a,b);
    return(0);
}

The output from this program is:

before a=3, b=5
after a=3, b=5

Because of pass-by-value, there is no change to main’s variables.

Pass-by-reference Using Pointers

Instead, we need to pass a “reference” to the two integers to be interchanged, so that the function swap() is actually dealing with the original variables, rather than new copies of their values:

#include <stdio.h>

void swap(int *ip, int *jp)
{
    int temp;

    temp = *ip;
    *ip = *jp;
    *jp = temp;
}

int main()
{
    int a=3, b=5;

    printf("before a=%d, b=%d\n",a,b);
    swap(&a, &b);
    printf("after a=%d, b=%d\n",a,b);
    return(0);
}
The output from this program is:

before a=3, b=5
after a=5, b=3

Here we’ve introduced a bit more syntax (and, typically, it uses punctuation characters).

- The address operator, &, is used to determine the (run-time) memory address of a variable. Here we require the memory address of the variables a and b before passing these addresses to swap(). Notice that we are still using pass-by-value parameter passing, but that we are passing addresses on the run-time stack.

- The two asterisks in swap()’s formal definition indicate that the variables ip and jp are pointer variables, rather than just “simple” variables. It is typical in C programs to append p or ptr to a variable’s name to indicate that it’s a pointer.

- The asterisks always placed in front of ip and jp in function swap() indicate that we wish to dereference these variables. Instead of using the contents of these variables (which are “meaningless” memory addresses) we wish to use the values that they point to. Notice that we may dereference variables on “both sides” of an assignment expression.

Pointers To Arrays And Character Strings

One confusing point in C is the synonymous use of arrays, character strings, and pointers. The name of an array in C is actually the memory address of the array’s first element. Thus the following two assignment statements are the same, and the first is the more commonly used:

```
char buffer[BUFSIZ], *ptr;

ptr = buffer;
ptr = &buffer[0];
```

Using cdecl:

```
-> explain char *ptr
  declare ptr as pointer to char
```

Remember that C’s character strings are simply a contiguous series of characters (which, by convention, are terminated by a NULL character), so we can consider strings to be arrays too. Strings may therefore be accessed through pointers (you may wish to consider a string’s first character as being stored at the memory address of the array of characters). We can thus write:

```
int n;
char *hex_values = "0123456789abcdef";

n = hex_values[ ...expression... ];
```
or (don’t do this!)

\[
n = "0123456789abcdef"[ \ldots \text{expression} \ldots ];
\]

We often see character pointers (to strings), and character arrays (with assumed terminating NULL characters) used interchangeably:

```c
int my_strlen(char *str)
{
    int len = 0;
    while ( str[len] /* != '\0' */ )
        ++len;
    return (len);
}
```

**Pointer Arithmetic**

Another confusing facility in C is the use of pointer arithmetic, by which we may advance a pointer to successive memory locations at run-time. It would make little sense to be able to “point anywhere” into memory, so C automatically adjusts a pointer (forwards or backwards) by values that are multiples of the size of the base types (or user-defined structures) to which it points.

We specify pointer arithmetic in the same way we specify numeric arithmetic, using $+$, $-$, and pre- and post-increment and decrement operators (multiplication and division make little sense). We can thus traverse an array with pointer arithmetic:

```c
int my_strlen(char *str)
{
    int len = 0;
    while ( *str /* != '\0' */ ) {
        ++len;
        ++str;
    }
    return (len);
}
```

Note that we are simply “moving the pointer along”. We are not modifying what it points to, simply accessing adjacent memory locations until we reach one containing the NULL character. This example is a little simple, because the character pointer will only be advanced one memory location (one byte) at a time, as a character is one byte long. Consider the five equivalent loops:
int sum_array(int *values, int n)
{
    int  i, *ip;
    int  sum = 0;

    for (sum=0, i=0 ; i<n ; ++i)
        sum += values[i];

    for (sum=0, i=0 ; i<n ; ++i)
        sum += *(values+i);

    for (sum=0, ip=values; ip<&values[n] ; ++ip)
        sum += *ip;

    for (sum=0, i=0 ; i<n ; ++i) {
        sum += *values;
        ++values;
    }

    for (sum=0, i=0 ; i<n ; ++i)
        sum += *values++;

    return (sum);
}

Unfortunately, we frequently see an excessive use of pointer arithmetic in C, with programmers trying to be too smart to speed up their programs. For example:

char *my_strcpy(char *dest, char *src)
{
    char   *d = dest;

    while (*dest++ = *src++ )
        return (d);
}

With code such as this, in which we are trying to copy all characters from src to dest until we reach NULL, we always have in the back of our minds the concern as to whether the NULL character is in fact copied.
Sorting An Array Of Values

A frequently required operation is to sort an array of, say, integers or characters. The standard C library provides a generic function named `qsort()` to help with this, but we must write a pointer-based function to perform the comparison of the array’s elements:

```c
#include <stdlib.h>

#define N 100

int compare(const int *ip, const int *jp)
{
    return (*ip - *jp);
}

int main()
{
    int i;
    int values[N];
    srand( getpid() );
    for (i=0 ; i<N ; i++)
        values[i] = random();
    qsort((void *) values, (size_t)N, sizeof (values[0]), compare);
    ...
    return (0);
}
```
Dynamic Memory Allocation

The function `malloc()` returns a requested number of bytes from the operating system’s heap. If insufficient memory is available, `malloc()` returns `NULL`. When we are finished using the space returned by `malloc()`, the program should return it to the heap with a call to `free()`. If a process continues to `malloc()` memory and fails to `free()` it, the process may “run out of memory”, and terminate ungracefully.

Unlike Java, C has no garbage collection of heap objects, and so programs must be careful about deallocating memory that is no longer required.

Consider the following example which allocates space for a new copy of a given string. This is very similar to the standard function named `strdup()`:

```c
char *newstr(const char *s)
{
    void *malloc(unsigned int nbytes);
    char *p;

    if ( (p=(char *)malloc(strlen(s)+1)) == NULL ) {
        fprintf(stderr,"out of memory!\n");
        exit(1);
    }

    strcpy(p,s);
    return (p);
}
```

`malloc()` is also frequently used to allocate memory for structures.

```c
#define NEW(t) (t *)malloc(sizeof (t))

struct l {
    char *line;
    struct l *next;
} ;

struct l *hd = (struct l *) malloc(sizeof (struct l) );

fgets(buf, MAX, fp);
while ( !feof(fp) ) {
    p = NEW(struct l);
    p->line = newstr(buf);
    p->next = hd;
    hd = p;
    fgets(buf,MAX,fp);
}
```
C Projects Developed In Multiple Files

Just as programs should be divided into functions and procedures, C programs themselves should be partitioned into files. Then small changes to one file do not require all files to be recompiled.

Assume that a project is partitioned into the four files proj.c, data.c, io.c, calc.c. Each file depends on a common header file proj.h. In the header file:

```c
#include <stdio.h>
#include <ctype.h>

#define FALSE 0
#define TRUE 1
#define MAX 80
#define NEW(t) ((t *)malloc(sizeof(t))

struct l {
    char *l;
    struct l *next;
};

extern struct l *hd;
extern char buf[MAX];
extern int linecount;
```

In the file data.c:

```c
#include "proj.h"

struct l *hd;
char buf[MAX];
int linecount;
```
In the other files :

```c
#include "proj.h"

int main(int argc, char *argv[])
{
    ... process arguments

    openfiles();    /* in io.c */
    doit();         /* in calc.c */
    exit(0);
}
```

**Compiling Multiple Files Using *make***

The program *make* maintains up-to-date versions of programs that result from a sequence of operations on a set of files.

*make* reads specifications from either *makefile* or *Makefile* and performs the actions associated with rules if indicated files are “out of date”. Basically:

```plaintext
if (files on which a certain file depends)
    i) do not exist, or
    ii) are not up-to-date
then
    create an up-to-date version;
```

*make* operates over rules and operations recursively and will abort if it cannot create an up-to-date file on which another file depends.

For the case of our multi-file project earlier :

```bash
proj : proj.o data.o io.o calc.o
    gcc -o proj proj.o data.o io.o calc.o

proj.o : proj.c proj.h
    gcc -c proj.c

data.o : data.c proj.h
    gcc -c data.c

io.o : io.c proj.h
    gcc -c io.c

calc.o : calc.c proj.h
    gcc -c calc.c
```
make also supports simple variable and macro substitutions:

NAME  = proj
OBJ   = proj.o data.o io.o calc.o
CC    = gcc
CFLAGS = -O

$(NAME) : $(OBJ)
  $(CC) -o $(NAME) $(OBJ)

proj.o : proj.c proj.h
  $(CC) $(CFLAGS) -c proj.c

data.o : data.c proj.h
  $(CC) $(CFLAGS) -c data.c

io.o : io.c proj.h
  $(CC) $(CFLAGS) -c io.c

calc.o : calc.c proj.h
  $(CC) $(CFLAGS) -c calc.c

clean :
  rm -f $(OBJ) core