Chapter 5
Pointers and Arrays

A pointer is just a variable with its value being an address of another variable. It is similar to the concept of reference in Java.

Pointers are used a lot in C programming because 1) sometimes, it is the only way to get things done; 2) they could lead to a more compact and/or efficient representation; and 3) pointers and arrays are closely connected.

On the negative side, it is not straightforward; and, if not used carefully, it could lead to quite some mess. 😞
Pointers and addresses

The memory in a typical machine can be represented as an array of consecutively numbered, or addressed, memory cells that may be manipulated individually or collectively as in arrays.

Any byte in such a memory can represent a character in char type, or a pair of bytes a value in short int type, or four adjacent types can be a value in the type int in Turing.

A pointer then can hold an address of any of the aforementioned group of bytes. In particular, if c is of char type, then we can declare cp as a pointer that points to c as follows:

```c
char c;
char *cp=&c;
```

The last line gets &c, the address of c, and put it into the pointer cp.
More examples

For example, *py, a pointer to an integer, can be declared as follows:

```c
int * py;
```

When the program needs to actually use the variable, we have to use `new` to ask for the space:

```c
py = new int;
```

Then, we can work with *py, the integer type variable pointed by py, e.g.,

```c
*py = 10;
```

These three steps can also be combined, e.g.,

```c
int * py = new int (10);
```
Thus, ‘&’ gives the address of a variable, and ‘*’ is the dereferencing operator which, when applied to a pointer, it accesses the object being pointed by that pointer, or “the stuff being pointed by that pointer”.

```c
int x=1, y=2, z[10];
int *ip; /* ip is a pointer to an integer */
ip=&x; /* ip points to x */
y=*ip; /* y is now assigned a value of 1 */
*ip = 0; /* x is now assigned the value of 0 */
ip = &z[0]; /* ip now points to the first element of z */
ip = z; // We will see....
```

Similarly, in

double *dp, atof(char *);

says that dp is a pointer for a double type quantity, and atof, with a pointer to char type of stuff, really a string, as an input, will send out a double quantity.
A few points

1. If \( \text{ip} \) points to the integer \( x \), then \( \ast \text{ip} \) can occur anywhere \( x \) could. For example

\[
\ast \text{ip} = \ast \text{ip} + 10;
\]

increments \( \ast \text{ip} \), i.e., the current value kept in the variable being pointed to by \( \text{ip} \), by 10.

**Question:** What does \( y = \ast \text{ip} + 1 \); do? How about \( y = \ast (\text{ip} + 1) \);

On the other hand, \( \ast \text{ip} += 1 \), \( +++ \text{ip} \) and \( (\ast \text{ip})++ \) all add a 1 to whatever \( \text{ip} \) points to.

**Question:** What does \( \ast \text{ip}++ \) do? How about \( \ast (\text{ip}++) \)?
Call by reference

When variables are passed as arguments to a function, only their values are copied into the formal parameters.

Once the copy is done, all the relationship between the actual parameters and the formal ones are cut. In particular, the values of those calling variables will not be changed by the called environment. This is what we mean by \textit{call by value}. We use this quite a bit in \textit{Java}.

To change the values of such calling variables within a function, we have to follow the approach of \textit{call by references}, which can be realized by using the addresses of the variables, such as the pointers.
Pointers as arguments

In coding the sorting algorithms, we used a swap method to exchange the values of two variables. Will the following version work?

```c
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}
```

**Question:** What will the following program print out?

```c
main(void){
    int x = 1, y = 2;
    swap(x, y);
    printf("%d %d\n", x, y);
}
```

**Question:** How could we fix it?

**A short answer:** Use pointers.
A longer one

Using the ideas of pointers, we can come up with another example of a swap function as follows:

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

**Question:** What will the following program print out?

```c
void main(void){
    int x = 1, y = 2;

    printf("Before the swap %d %d\n", x, y);
    swap(&x, &y);
    printf("After the swap %d %d\n", x, y);
}
```
Cut it open...

1. swap(int* px, int* py)

   int* px=&x; int* py=&y;

   Now, px and py “point” to a and b.

2. int temp = *px;

   The value of a, pointed by px, goes into temp

3. *px = *py;

   The value of y, pointed by py, goes to x.

4. *py = temp;

   The value of temp goes to y, the one pointed by py.

It turns out that this is the way referred to as “call by reference” in C. In other words, the reference mechanism in Java does not work in C, 😞 but it does in C++. Stay tuned! 😊
Pointers and arrays

There is a strong relationship between pointers and arrays in the sense that the name of an array is a “pointer” to the first element of the same array.

```c
int a[10];
int *pa=&a[0];
```

The address of `a[0]`, the very first element of `a`, is passed to `pa`, a pointer variable.

Thus, anything that can be done with the array structure can also be done with pointers, and often faster.

Continue with the above example,

```c
int x=*pa;
```

will pass the value of `a[0]` to `x`.

Moreover, the expressions `*(pa+1)`, `*(pa+2)` will refer to `a[1]` and `a[2]`, respectively.
Remember this stuff?

Given the following declaration:

```c
#define N 100
int a[N], i, *p, sum=0;
```

Assume the system assigns 300 as the base address of the array `a`, and that memory bytes numbered 300, 304, 308, ..., 696 are assigned as the addresses of `a[0]`, `a[1]`, `a[2]`, `a[99]`, respectively, four bytes each for integers.

Then, the following segment

```c
for (p=a; p<&a[N]; ++p)
    sum+=*p;
```

will add up all the elements in this array, which is the same as

```c
for (i=0; i<N; ++i)
    sum+=*(a+i);
```

Both of these is certainly the same as the following that we did in Java:

```java
for (i=0; i<N; ++i)
    sum+=a[i];
```
A few more points

1. Since the constant \( a \) holds the location of its first element, \( \text{pa} = \&a[0] \); is the same as \( \text{pa} = a \);

   In fact, when C sees \( a[i] \), it immediately converts it to \( *(a+i) \). Expressions such as \( \&a[i] \) and \( (a+i) \) are equivalent.

2. Notice that a pointer is a variable, but an array name is not, which is fixed by the compiler. Thus, this sequence is legal

   \[ \text{pa} = a; \]
   \[ \text{pa}++; \]

   but none of the following is:

   \[ a = \text{pa}; \]
   \[ a++; \]
An example

The following function, defined in string.h, is to find out the length of a string.

```c
int strlen(char *s){
    int n;
    for(n=0; *s!='\0'; s++) n++;
    return n;
}
```

All the following calls work:

```c
strlen("hello, world."); /* string constant */
strlen(a); /* char a[100]; */
char * ptr=a;
strlen(ptr); /* char *ptr; */
```

**Question:** Does the length include ‘\0’?
Shall we test it out?

/home/PLYMOUTH/zshen > more strlen.c
#include <stdio.h>
#include <string.h>

int strlen1(char *);

main(){
    char s[20]="Hello, World."
    char *sPtr=s;

    printf("%d\n", strlen1(s) );
    printf("%d\n", strlen1("Hello, World.") );
    printf("%d\n", strlen1(sPtr) );
}

int strlen1(char *s){
    int n;

    for(n=0; *s!='\0'; s++) n++;

    return n;
}

/home/PLYMOUTH/zshen > cc strlen.c
/home/PLYMOUTH/zshen > ./a.out
13
13
13
Address arithmetic

If $p$ is a pointer to some element of an array, then $p++$ increments $p$ to point to the next element, and $p+=i$ points to the element $i$ positions beyond the current one as pointed by $p$.

Such integration of pointers, arrays and address arithmetic is a distinguished strength of the C language.

Let’s further demonstrate this phenomenon by writing a simple space allocator, which has two services. The first one, `alloc(n)` returns a pointer $p$ to $n$ consecutive character positions; while the second, `afree(p)`, releases the storage with its first position being pointed by $p$.

The system provides more general allocation services `malloc()` and `free(p)`. Check out the examples as given in the Course page.
An implementation

An easy implementation is to prepare a large chunk of memory allocbuf and the function alloc(n) will simply give out n pieces of space from allocbuf.

It is certainly a good idea to treat this memory private to both alloc and afree. How do we do it? 😊

To know how much space has been allocated, we also use another pointer allocp, that points to the next available element.

When alloc(n) is called, it checks to see if there is still this much space available in the buffer, and, if so, the value of allocp is returned, which is then advanced for n positions. Otherwise, alloc(n) returns 0.
Let’s look at the code

#define ALLOCSIZE 10000
/* global but private */
static char allocbuf[ALLOCSIZE];
static char *allocp=allocbuf;

char *alloc(int n){
    if(allocbuf+ALLOCSIZE-allocp>=n){
        //allocp points to where the next
        //assignment can be made
        allocp+=n;
        return allocp-n;
    }
    else return 0;
}

void afree(char *p){ //If p is within the range
    if((p>=allocbuf)&&(p<allocbuf+ALLOCSIZE))
        allocp=p;
}

Question: Is this implementation practical, or even correct? 😊
A few points

1. If \( p \) and \( q \) point to members of the same array, i.e., the stuf of the same type, then they can be compared with such comparators as ==, !=, <, >= etc. For example, \( p<q \) means if \( p \) points to an earlier location than \( q \) does.

2. We can add something to a pointer. But, \( p+n \) means the address of the \( n^{th} \) object beyond the point where \( p \) points to, but not just adds a value \( n \) to that of \( p \). Thus, the result depends on the nature of \( p \).

3. We can also do subtraction to a pointer.

```c
int strlen(char *s){
    char *p=s;
    while (*p!='\0')
        p++;
    return p-s;
}
```
Character pointers

As we discussed a few times, a string constant, e.g., "I am taking the C course" is an array of characters, ended with an invisible ‘\0’ so that a program knows where it ends. When such a string constant is passed into a function, the function is simply given a pointer to the very first character of this string.

Another place where such a constant is used is in

```c
char amessage[]="What is the difference?";
char * pmessage="What is the difference?";
```

**Question:** What is the difference between the following:

```c
char amessage[]="What is the difference?";
char * pmessage="What is the difference?";
```
An example

The function `strcpy(s, t)` copies the string `t` to `s`.

```c
void strcpy(char *s, char *t){
    int i=0;
    while((s[i]=t[i])!='\0')
        i++;
}
```

Another version could be

```c
void strcpy(char *s, char *t){
    while((*s=*t)!='\0'){
        s++; t++;
    }
}
```

We can do some further simplification:

```c
void strcpy(char *s, char *t){
    while((s++=*t++)!='\0');}
```

We can even do the following:

```c
void strcpy(char *s, char *t){
    while(*s++=*t++);
}
```

**Question:** Did we copy '\0'?
Another example

The Java method `strcmp(s, t)` compares two strings and returns negative, zero, or positive if `s` is lexicographically less than, equal to, or greater than `t`.

The C function behaves the same....

```c
int strcmp(char *s, char *t){
    int i;

    for(i=0; s[i]==t[i]; i++)
        if(s[i]=='\0')
            return 0;
    return s[i]-t[i];
}
```

This is what it should look like, using pointers.

```c
int strcmp(char *s, char *t){
    for( ; *s==*t; s++, t++)
        if(*s=='\0')
            return 0;
    return *s-*t;
}
```
A bit more

We can also use the following:

```
*--p; // ‘--’ has higher precedence.
```

which means an access to the object pointed by a decrement of $p$. In fact, we often use the following pair to do stack push and pop.

```
*p++=val; //(*p)++ or *(p++)?
val=*--p;
```

The first one places a value contained in $val$ to the box being pointed by $p$, then increments $p$, the stack top pointer; while the second decrements the pointer first, then places the value stored there to $val$.

*Check out the precedence list, as given in the Course page, as well as the Stack package as described in the previous chapter.*
Yet another...

```
#include <stdio.h>

main(){
    char sA[] = "I am ";
    char sB[] = "very very hungry!!";

    printf("String 1: %s\n", sA);
    printf("String 2: %s\n", sB);

    strCat(sA, sB);
    printf("Combined String: %s\n",sA);
}

strCat(char *s, char *t) {
    while(*s!='\0')
        s++;
    while('\0' != (*s = *t)) {
        s++; t++;
    }
}
```

```
/home/Plymouth/zshen > more strcat.c
/home/Plymouth/zshen > cc strcat.c
/home/Plymouth/zshen > ./a.out
String 1: I am
String 2: very very hungry!!
Combined String:  I am very very hungry!!
```
Labwork

1. Implement `strend(char * s, char *t)`, which returns 1 if `t` occurs at the end of `s`, and 0 otherwise.

So, `strend("I don’t want to fail", "HCI")` returns 0, but `strend("I want to pass", "pass")` returns 1.

2. Implement `strNcmp(char * s, char * t, int n)`, which compares the first `n` characters of `s` and `t` and sends back a 1 if they match, and a 0 otherwise.

For example, `strNcmp("fail", "fall", 3)` returns 0, while `strNcmp("fail", "fail", 3)` returns 1.

Send in a driver for each of the two functions.
Array of pointers

Since an array is a list of something of a certain type, and pointers are variables of certain types themselves, we can put them into an array as well, i.e., an array of pointers.

Let’s check out an algorithm that sorts a collection of strings, using an array of pointers each of which points to a character string.

The algorithm essentially consists of three steps:

read all the lines of input
sort them
print them out in order
The code for the program

```
/home/PLYMOUTH/zshen > more main.c
#include <stdio.h>
#include <string.h>
#include "myFile.h"

#define MAXLINES 5000

char *lineptr[MAXLINES];

main(){
    int nlines;

    if((nlines=readlines(lineptr, MAXLINES))>=0){
        printf("\nThis is what we just read in:\n\n");
        writelines(lineptr, nlines);
        printf("\nNow sorting begins:\n\n");
        qsort(lineptr, 0, nlines-1);
        //insertsort(lineptr, nlines);
        writelines(lineptr, nlines);
        return 0;
    }
    else {
        printf("error: input too big to sort\n");
        return 1;
    }
}
```
/home/PLYMOUTH/zshen > more inOut.c
#include <stdio.h>
#include <string.h>
#include "myFile.h"
define MAXLEN 1000
#define MAXLINES 5000

extern char *lineptr[MAXLINES];

int readlines(char *lineptr[], int maxlines){
  int len, nlines;
  char *p, line[MAXLEN];

  nlines=0;
  while ((len=getline1(line, MAXLEN))>0)
    if(nlines>=maxlines||(p=alloc(len))==NULL)
      return -1;
    else {
      line[len-1]='\0'; /* It was '
'.
      strcpy(p, line);
      lineptr[nlines++]=p;
    }
  return nlines;
}

void writelines(char *lineptr[], int nlines){
  int i;
  for(i=0; i<nlines; i++)
    printf("%s\n", lineptr[i]);
}
The qsort sorts the list $v$ into order. For details, see p. 87 of the textbook.
Insertion sort

How about another one?

```
/home/PLYMOUTH/zshen > more isort.c
void insertsort(char *v[], int length){

    int n=length;
    int i, j;

    for (j = 1; j < n; j++) {
        char * temp = v[j];
        // Insert array[j] into array[0..j-1].
        i = j-1;

        while (i >= 0 && strcmp(v[i], temp) > 0) {
            v[i+1] = v[i];
            i--;
        } //while
        v[i+1] = temp;
    } //for

/home/PLYMOUTH/zshen > more myFile.h
char *alloc(int);
int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void insertsort(char *v[], int length);
void qsort(char *lineptr[], int left, int right);
```
Check it out

/home/PLYMOUTH/zshen > cc alloc.c inOut.c quicksort.c main.c
/home/PLYMOUTH/zshen > more testSort.txt
aerqwertwe
sdafdasfgd
between
qweetewr
cctest
/home/PLYMOUTH/zshen > ./a.out < testSort.txt

This is what we just read in:

aerqwertwe
sdafdasfgd
between
qweetewr
cctest

Now sorting begins:

aerqwertwe
between
cctest
qweetewr
sdafdasfgd
Just a bit of multi-d array

C provides rectangular multi-dimensional arrays, although not as much used as arrays of pointers.

For example, there are twelve months for each year, and each month has different number of days. In particular, the number of days in February is either 28 or 29 depending on if that year is leap or not. Last year is but not this one... .

Thus, the basic structure for a date conversion program could be the following:

```
static char daytab[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```
Initialize a pointer array

We use an example to demonstrate how to initialize an array of pointers.

Let's write one function that sends back the name of a month.

```c
char *month_name(int n){
    static char *name[]={
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n<1||n>12)? name[0]:name[n];
}
```
I am confused...

**Question:** What is the difference between the following two lines?

```c
int a[10][20];
int *b[10];
```

Although `a[3][4]` and `b[3][4]` could refer to a single integer, they are fundamentally different.

When the system sees `a[10][20]`, it will set aside a $10 \times 20$ block for the array `a`; and will use a formula

$$20 \times row + col$$

to figure out the location of `a[row][col]` in such a block.

This order is called “row major”, which is adopted by most of the “popular” programming languages, except Fortran, OpenGL, and Matlab.

*Check out the relevant link on the course page.*
For $b$, the only thing the system will do is to give a block of 10 pointer worth space, and will not allocate anything else. Assume later on, each pointer does point to a 20 element int elements, then the total amount of space set aside is 200 int type elements plus 10 cells for pointers.

The advantage to use $b$ is that we don’t need to allocate arrays of fixed length to all the 10 cells; and we don’t need to always commit this much space all the time.

The most frequently used arrays of pointers are certainly for character strings, which often have different length.

char *name=\{"Illegal month", "Jan", "Feb", "March"\};

**Question:** What is the structural difference between the above line and the following:

char aname[][15]=\{"Illegal month", "Jan", "Feb", "March"\};
Command-line arguments

This is an important application of arrays of character strings, by providing a way of passing in arguments when executing a program.

They always come with a Java program, although we usually don’t use them in G-rated programs. 😊

public static void main(String args[])

When in need, main is called with two arguments, argc, providing the number of arguments, and argv, an array of pointers pointing to a list of arguments as character strings.

For example, when calling the following

> echo hello, world

it prints out all the arguments, separated by blanks, in a single line, i.e.,

hello, world
How does it do it?

By convention, argv[0] is the name of the invoked program, so argc is at least 1, where there might be something else.

The first version of echo treats argv as an array of character pointers.

```
#include <stdio.h>
main(int argc, char *argv[]){
  int i;
  for(i=1; i<argc; i++)
    printf("%s%s", argv[i], (i<argc-1)? " ": "");
  printf("\n");
  return 0; //The implicit return type of int 
}
```

```
/home/PLYMOUTH/zshen > ./a.out Hello, World.
Hello, World.
```

In the above invocation of echo hello, world, argc=3, and argv contains three elements: "echo", "hello," and "world.".
Treat it as what it is...

Since `argv` is really a pointer, we can work with it directly, rather than an index `i`.

```
/home/PLYMOUTH/zshen > more echo2.c
#include <stdio.h>

main(int argc, char *argv[]){
    //argc is the number of arguments
    //argc-1 is the highest index
    while(--argc>0)
        //We don’t want to print out the program name.
        printf("%s%s", *++argv, (argc>1)? " " : "");
    printf("\n");
    return 0;
}
```

```
/home/PLYMOUTH/zshen > cc echo2.c
/home/PLYMOUTH/zshen > ./a.out Hello, World.
Hello, World.
```
Another example

Below is another version of the pattern finding program, with a flexible *argument*.

```
/home/PLYMOUTH/zshen > more matchArgument.c
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline1(char *line, int max);
main(int argc, char *argv[]){
    char line[MAXLINE];
    int found=0;

    if(argc!=2)//Handy reminder....
        printf("Usage: a.out pattern\n");
    else while(getline1(line, MAXLINE)>0)
        if(strstr(line, argv[1])!=NULL){
            printf("%s", line);
            found++;
        }
    return found;}
```

/home/PLYMOUTH/zshen > cc matchArgument.c
/home/PLYMOUTH/zshen > ./a.out ould < test.txt
Ah love! could you and I with Fate conspire
Would not we shatter it to bite -- and then
Re-mould it nearer to the Heart’s Desire!

*Check out strstr on the course page.*