Pointers

1 Introduction

A pointer is a variable whose value is the address of another variable [Pet97]. Hence we think of them as 'pointing' to another variable. Knowing the specific address of a variable is typically not what we are interested in. Instead, we can use the pointer to access the value of the variable whose address is stored in the pointer variable.

Pointer variables have their own type, which should match the type of variable that they point to. For example, a pointer that contains the address of an int will be of type pointer-to-int, a pointer that contains the address of a double will be of type pointer-to-double, and so forth. This becomes relevant when we declare pointer variables, which must be declared just as variables of other types of variables are declared. For all of our examples, we assume we are on a machine in which a variable of type char is one byte, a variable of type int is four bytes, a variable of type double is eight bytes, and an address is four bytes.

2 Pointers to Variables

In order to get the address of a variable, we precede it with an ampersand (i.e., &), also known as the address operator. To declare a pointer variable, we precede the variable name by an asterisk (i.e., *). For example, the following lines of code declare and initialize a variable x of type int, declare a variable ptr of type pointer-to-int (or int-star), and assign the address of x to ptr:

Example 1

```c
int x = 5;
int* ptr;
ptr = &x;
```

Just as other types of variables can be declared and initialized in the same line of code, we can declare and initialize pointer variables. Therefore, I could have written Example 1 like this:

Example 2

```c
int x = 5;
int* ptr = &x;
```

In the previous examples, when I declared the pointer variable the type was int* and the name of the variable was ptr. Since the value of ptr is the address of the variable x, when I use the name ptr I get the address. For example, compiling and running the program in Example 3

Example 3
```c
#include <stdio.h>

int main(void)
{
    int x = 5;
    int* ptr = &x;
    printf("the value of ptr is %p\n", ptr);
}

Produces

the value of ptr is 0xbf893f10

which is an address in hexadecimal. Notice the format specifier we used in the printf() statement in Example 3 is %p, which indicates that we are printing the content of a pointer.

As I stated in Section 1, we typically are not interested in the specific address in memory where a value is stored and, in fact, this value is determined at run time anyway. In most cases we would like to use the pointer to access the value at the address stored in the pointer. To do this, we precede the pointer variable with an asterisk, also known as the indirection operator. By doing so, we can use the pointer in place of the variable it points to; see Example 4.

Example 4
#include <stdio.h>
int main(void)
{
    int x = 5;
    int y = 20;
    int* ptr = &x;

    printf("x is %d, *ptr is %d\n", x, *ptr);
    y += *ptr; /* line 11 */
    printf("y is now %d\n", y);
    (*ptr)++; /* line 14 */
    printf("x is now %d, *ptr is now %d\n", x, *ptr);
}

produces
x is 5, *ptr is 5
y is now 25
x is now 6, *ptr is now 6
On line 11 of the program in Example 4, I am using the pointer to add the value of x to the value of y. On line 14 I use the pointer to increment the value of x. Notice that when doing so, I have enclosed *ptr in parentheses. The indirection operator and the increment operator (i.e., ++) have the same level of precedence, but these operators are evaluated from right to left. Therefore, the statement

\texttt{(*ptr)++;}\n
increments the value of the variable pointed to by ptr, which is what I wanted here, while

\texttt{*ptr++;}\n
would increment the content of ptr (i.e., change the value of the address it points to) and then get the value at the new address. This is legal, and useful in some instances, but not what I wanted in this case.

3 Pointers as Function Parameters

It is common in C to call a function that returns a value, which we store in a local variable. For example,

\texttt{y = someFunction(x);}\n
saves the value returned by someFunction in the variable y. C does not allow a function to return multiple values. The way around this is to pass the addresses of local variables to the function, which then manipulates the values of the local variables via pointers as shown in Example 5.

Example 5

\texttt{#include <stdio.h>}\n
\texttt{void swap(int*, int*);}\n
\texttt{int main(void)}\n
\texttt{\{}\n
\texttt{int x = 5;}\n
\texttt{int y = 8;}\n
\texttt{swap(&x, &y);}\n
\texttt{printf("x is %d, y is %d\n", x, y);}\n
\texttt{\}}\n
\texttt{void swap(int* a, int* b)}\n
\texttt{\{}\n
\texttt{int temp = *a;}\n
\texttt{*a = *b;}\n
\texttt{*b = temp;}\n
\texttt{\}}\n
produces\n
x is 8, y is 5
Notice that in the function call, we use the address operator to get the addresses of variables \( x \) and \( y \). In our function definition, we declare two variables of type pointer-to-int, so we are essentially doing this:
```
int* a = &x;
int* b = &y;
```
We then use \( \ast a \) and \( \ast b \) in the function just as we would had we performed the swap in \texttt{main()} instead.

4 Pointers to One-Dimensional Arrays

We can have pointers to arrays. To create a pointer to an array, we can do something like this:

Example 6
```
int data[4] = {1, 2, 3, 4};
int* ptr = data;
```
Notice in this case that I did not precede the name of the array, \texttt{data}, by an ampersand.
This is because using the name of the array gives us the address of the beginning of the array, which we can see in Example 7.

Example 7
```
#include <stdio.h>

int main(void)
{
    int data[4] = {1, 2, 3, 4};
    int* ptr = data;

    printf("the location of data is %p\n", data);
    printf("the value of ptr is %p\n", ptr);
}
```
produces
```
the location of data is 0xbfcc3c14
the value of ptr is 0xbfcc3c14
```

When I declared a pointer to a variable of type int, I could use or change the value of that variable by prefixing the pointer with an asterisk. I can do the same with a pointer to an array, as shown in Example 8.

Example 8
```
#include <stdio.h>
```
int main(void)
{
    int data[4] = {8, 9, 10, 11};
    int* ptr = data;
    int i;

    printf("data begins at %p\n", data);
    printf("Element Value Address\n");
    printf("-----------------------\n");

    for(i = 0; i < 4; i++)
    {
        printf(" %d %2d %p\n", i, *ptr, ptr);
        ptr++;
    }
}

produces

data begins at 0xbfd03700

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>0xbfd03700</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>0xbfd03704</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0xbfd03708</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>0xbfd0370c</td>
</tr>
</tbody>
</table>

To understand how this program works, we need to discuss two other things: how array elements are stored in memory and pointer arithmetic. When an array is declared, memory is allocated for it such that all of the variables will be in contiguous memory locations. That is, they are right next to each other in memory and, as was demonstrated by this program, we can just increment the address stored in our pointer so that it points to the address of the next element.

We can see from the output of the program that the memory addresses of the elements next to each other differ by 4, representing the four bytes that each int occupies in memory, yet in the program the pointer value was incremented using the increment operator. When we use this to increment the value of an int, it adds 1 to the int so how does this allow us to move from one address to the next? When we use the increment operator on a pointer, it uses pointer arithmetic to determine how much to add to the address. Pointer arithmetic uses the size of the type of variable that the pointer point to determine how much to change the address. In this example the variables stored in the array are ints, each of which takes four bytes. By incrementing the pointer by 1, we are really adding the size of one int to the address stored in the pointer.

We're not limited to incrementing an address by just 1; we can increment by other amounts and even decrement it as well. In Example 9 the value of the pointer is
initially set to the address of the beginning of the array, which is also the address of the first element. By incrementing the address by the size of three ints, we get the address of the last element. By decrementing this address by the size of two ints, we get the address of the second element.

Example 9

```c
#include <stdio.h>

int main(void)
{
    int data[4] = {8, 9, 10, 11};
    int* ptr = data;

    printf("address of %d is %p\n", *ptr, ptr);

    ptr += 3;
    printf("address of %d is %p\n", *ptr, ptr);

    ptr = ptr - 2;
    printf("address of %d is %p\n", *ptr, ptr);
}
```

produces
address of 8 is 0xbfd67cb4
address of 11 is 0xbfd67cc0
address of 9 is 0xbfd67cb8

While there is nothing wrong with creating a pointer to the array and using it to access the array elements, we can use the fact that the array name evaluates to the address of the beginning of the array. However, since an array has multiple elements, we need to include some additional information to reference specific elements of the array. For this we need to include an offset from the beginning of the array (i.e., the base) to the specific element. To get the address of any element of the array, we just add the number of elements from base to the element in question:

```
    array_name + offset
```

To get the value at this address, we use the indirection operator:

```
    *(array_name + offset)
```

Example 10

```c
#include <stdio.h>

int main(void)
{
}
```
```c
{ int data[4] = {8, 9, 10, 11};
 printf("address of %2d is %p\n", *(data + 0), data + 0);
 printf("address of %2d is %p\n", *(data + 3), data + 3);
}
```

produces

address of 8 is 0xbf90cbd8
address of 11 is 0xbf90cbe4

In Example 10, when the offset was zero I could have left it out and just used data to get the address of the first element of the array and *data to get its value. How does this approach, referred to as pointer notation, compare to using subscript notation to access array elements? The offset and the subscript are both equal to the index of the array element; see Example 11.

**Example 11**

```c
#include <stdio.h>

int main(void)
{
    int data[4] = {8, 9, 10, 11};
    int i;
    for (i = 0; i < 4; i++)
        printf("%2d, %2d\n", *(data + i), data[i]);
}
```

produces

8, 8
9, 9
10, 10
11, 11

### 5 Pointers to Two-Dimensional Arrays

Using pointer notation to access elements of a two-dimensional (2D) array is much like accessing a one-dimensional array. The difference is that now we must account for the rows as well as the columns when considering offsets. The form for getting the address of a specific element of the array is

*(array_name + row) + col

To get the value at this address, we use

*(*(array_name + row) + col)

This is equivalent to array_name[row][col] if we are using subscript notation.

Let’s break down the form for using pointer notation to access the elements of a 2D array. It is helpful to remember that a 2D array is an array of arrays. If the
dimensions of the array are $R \times C$, then there are $R$ rows, each of which is an array. Each of these $R$ arrays has $C$ elements of the type the array was declared to store, for example int.

The part of the form

*$(array\_name + row)$

produces the address of a row. For example, if row = 2 and the array type is int, then *$(array\_name + 2)$ is the address of the beginning of the 3rd row. Pointer arithmetic does the work for us of adding the size of two rows of ints to the base address. If we then include the offset for the column of the array element,

*$(array\_name + row) + col$

we get the address of the specific element of the array. To get the value of this element, we add another indirection operator:

*$(*(array\_name + row) + col)$

If either the row or column values are zero, we can leave them out but must be careful that we keep whatever parentheses are necessary to maintain the appropriate relationships between the indirection operators and the offsets. For example,

*$(*(data + 2) + 0)$ $\iff$ **$(data + 2)$ $\iff$ data[2][0]

and

*$(*(data + 0) + 3)$ $\iff$ *$(data + 3)$ $\iff$ data[0][3]

If both the row and column values are zero, we can leave both out to get

**$data$

Example 12

```c
#include <stdio.h>

int main(void)
{
    int data[2][3] = {{8, 9, 10},
                      {11, 12, 13}};

    printf("%p\n", *(data + 1));
    printf("%2d\n", **(data + 1));
    printf("%p\n", *(data + 1) + 2);
    printf("%2d\n", *(data + 1) + 2);
}
```

produces

```
0xbf96cf74
11
0xbf96cf7c
13
```

In Example 12, I worked with the array name instead of creating a pointer to the array. How would this work?
Example 13
#include <stdio.h>

int main(void)
{
    int i;
    int data[4][3]= {{ 1, 2, 3},
        { 4, 5, 6},
        { 7, 8, 9},
        {10, 11, 12} };
/* (*rowptr)[3] is a pointer to the first row (an array) of data.
*rowptr is the address of one of the rows. to get the value of a
specific element on the row, we prefix with * again */
int (*rowptr)[3] = data;

/* use offsets to the base address to print the second column */
for(i = 0; i < 4; i++)
    printf("%p %2d\n", *(rowptr + i) + 1, *(rowptr + i) + 1);
printf("\n");
for(i = 0; i < 4; i++)
{
    printf("%p %2d\n", *rowptr + 1, *(*rowptr + 1));
    rowptr++;
}
}

produces
0xbfc95e6c 2
0xbfc95e78 5
0xbfc95e84 8
0xbfc95e90 11

0xbfc95e6c 2
0xbfc95e78 5
0xbfc95e84 8
0xbfc95e90 11

In order to understand Example 13, we need to remember that a 2D array is an
array of arrays. When we use just the name, data, we are referencing the first element
of the array which is another array (we can think of this as the first of the four arrays, or
rows, inside data). Therefore, in order to create a pointer to a row, we use
int (*rowptr)[3] = data;
which says that our variable, rowptr, is a pointer to an array that is the size of three ints.
To use this pointer, in the first for loop the row offset changes while the column offset is 1 in order to print the second column. In the second for loop, instead of using both row and column offsets the value of rowptr is incremented to the next row and then a column offset is added to get the second column.

Another way of thinking about this is that after incrementing the address in rowptr, to work with the current row the row offset will be zero. Therefore, the following statements are equivalent:

\[ *(rowptr + 1) \iff (rowptr + 0) + 1 \]

### 6 Pointers to Arrays as Function Parameters

When we have an array as a function parameter, we are already passing the address of the array to the function so there is no need to prefix with the address operator. Within the function, we access the array elements just as we did when creating pointers to an array within main().

**Example 14**

```c
#include <stdio.h>

void printArray(double*);

int main(void)
{
    double data[6] = {1.3, 7.9, 2.5, 8.0, 4.1, 9.4};
    printArray(data);
}

void printArray(double* ptr)
{
    double* endptr = ptr + 6;

    while(ptr < endptr)
    {
        printf("%p %5.1f\n", ptr, *ptr);
        ptr++;
    }
}
```

produces

```
0xbfbbee050 1.3
0xbfbbee058 7.9
0xbfbbee060 2.5
0xbfbbee068 8.0
0xbfbbee070 4.1
0xbfbbee078 9.4
```
In Example 14 we create an array of doubles and then use a function to print the array elements. What is the purpose of the following line?

double* endptr = ptr + 6;

This line creates a pointer to the address past the last array element. After printing the array element pointed to by ptr, ptr is incremented (again, using pointer arithmetic) to the next array element until it reaches an address outside the memory allocated for the array.

We can see from the output that the memory addresses are eight bytes apart, which is what we would expect if each double is eight bytes wide.

As in some earlier examples, instead of creating a pointer to navigate through the array I could have used pointer notation to reference each array element. The function declaration and definition remain the same, but inside the function I use offsets to access the specific array elements.

Example 15

#include <stdio.h>

void printArray(double*);

int main(void)
{
    double data[6] = {1.3, 7.9, 2.5, 8.0, 4.1, 9.4};
    printArray(data);
}

void printArray(double* ptr)
{
    int i;
    for(i = 0; i < 6; i++)
        printf("%p %5.1f\n", ptr + i, *(ptr + i));
}

produces
0xbf84f480 1.3
0xbf84f488 7.9
0xbf84f490 2.5
0xbf84f498 8.0
0xbf84f4a0 4.1
0xbf84f4a8 9.4

Why didn’t I use the array name as I would have had the for loop been in main? Remember that because of variable scope, the array name is unknown inside the function and therefore I needed to create some way to reference it.
To work with 2D arrays is similar to what we described in Section 5. Example 16 has two functions, each of which prints the first column of an array. printColumn increments a pointer to the row of the array. printColumn2 uses a row offset to get the row to work with.

Example 16

#include <stdio.h>

void printColumn(int (*rowptr)[3]);
void printColumn2(int (*rowptr)[3]);

int main(void)
{
    int data[4][3]= { { 1, 2, 3},
                       { 4, 5, 6},
                       { 7, 8, 9},
                       {10, 11, 12} };

    printColumn(data);
    printf("\n");
    printColumn2(data);
}

void printColumn(int (*rowptr)[3])
{
    /* (*rowptr)[3] is a pointer to the first row (an array) of data.
    *rowptr is the address of one of the rows. to get the value of a
    specific element on the row, we prefix with * again */

    int i;
    for(i = 0; i < 4; i++)
    {
        printf("%p %2d\n", rowptr, **rowptr);
        rowptr++;
    }
}

void printColumn2(int (*rowptr)[3])
{
    int i;
    for(i = 0; i < 4; i++)
        printf("%p %2d\n", *(rowptr + i), **(rowptr + i));
}
produces  
0xbfe30120 1  
0xbfe3012c 4  
0xbfe30138 7  
0xbfe30144 10  

0xbfe30120 1  
0xbfe3012c 4  
0xbfe30138 7  
0xbfe30144 10  

Notice that in printColumn2, to get the address of the current element I use  
*(rowptr + i)  
while to get its value I use  
**(rowptr + i)  
These two expressions demonstrate that in order to know how many asterisks to use  
we must know how many dimensions our array has. Knowing that data is 2D, it may be  
easier to understand these expressions if we add the column offset to each:  
*(rowptr + i) <==> (*(rowptr + i) + 0)  
and  
**(rowptr + i) <==> *(*(rowptr + i) + 0)  

7 Summary  
It’s normal when first learning to use pointers to be confused by when to use an  
asterisk and how many to use. The answer depends on what the pointer points to and  
how you are using it. In particular when working with array names, we need to keep in  
mind whether or not we have enough information to determine which array element to  
get the value of.  
When we create a pointer to an int, the concept of row and column offsets  
doesn’t make sense; the address of the element gives us everything we need to know.  
Therefore, if we have the following code segment  
int x = 7;  
int* ptr;  
ptr = &x;  
to change or use the value stored in x we can simply prefix the variable ptr with the  
indirection operator, *. Using *ptr is like using x.  
When we have a one-dimensional array, the name of the array is a reference to  
the beginning of the array. That alone is not enough to get the value of a specific array  
element other than the first element of the array. We must add an offset of how far the  
specific array element is from the beginning of the array. Once we have added the  
offset to the address of the beginning of the array, we can then prefix this address with  
the indirection operator to use the value. If we have  
int data[] = {1, 2, 3, 4};
Then data + 0, or just data, is the address of the first array element, data + 1 is the address of the second element, and so forth. To get the value of the second array element, we use *(data + 1).

When we wish to get the value of an element of a two-dimensional array, we have to not only consider how many columns the array element is from the beginning of the array, but also how many rows. Therefore we use two offsets. Let’s look at Example 17 to see how this works.

Example 17

```c
#include <stdio.h>

int main(void)
{
    int num[2][3] = { {25, 26, 27},
                     {28, 29, 30} };  
    int* numptr = &num[0][0]; /* get address of first element of the array */
    int i, j;

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 3; j++)
        {
            printf("%d %p", num[i][j], &num[i][j]);
            printf("\n");
        }
    }

    printf("\n");
    printf(" num is %p\n", num ); /* address of num array */
    printf(" *num is %p\n", *num ); /* address of num array */
    printf(" num + 1 is %p\n", (num + 1) ); /* address of second row */
    printf(" *(num + 1) is %p\n", *(num + 1) ); /* address of second row */
    printf(" *(num + 1) + 2 is %p\n", *(num + 1) + 2 ); /* address of second row, third column */
    printf("**(num + 1) + 2) is %d\n", *(num + 1) + 2 ); /* value of second row, third column */
    printf("\n");
    /* why only one star here? because numptr points to an int, not an array of ints */
    for(i = 0; i < 6; i++)
    {
        printf("%d ", *numptr++); /* this uses *numptr, then increments */
    }
    printf("\n");
}
```

produces
25  0012FF68  26  0012FF6C  27  0012FF70
28  0012FF74  29  0012FF78  30  0012FF7C

num is 0012FF68
*num is 0012FF68
num + 1 is 0012FF74
*(num + 1) is 0012FF74
*(num + 1) + 2 is 0012FF7C
*(*(num + 1) + 2) is 30

25  26  27  28  29  30

Let’s look at the individual parts of the expression for accessing the value of the last element of the array.
num <= address of beginning of array
*(num + 1) <= address of beginning of row 1 (since counting starts at 0)
*(num + 1) + 2 <= address of element at row 1, column 2
*(*(num + 1) + 2) <= value at row 1, column 2

By the time we get *(num + 1) + 2, we have added the necessary offsets from the base address to get to the last array element. At this point prefixing it with an indirection operator gives us the value of the last array element.

In this same example, we also created a variable, numptr. When we use it, we only use a single indirection operator to get the value of an array element. Why? The reason is because numptr is a pointer to an int and we initialized it to the first value of the array with

```c
int* numptr = &num[0][0];
```

Since num[0][0] is the first array element, prefixing it with the address operator gives us its address. This is really no different than in earlier examples when we created variables of type int and then created pointers to them. Prefixing numptr with a single asterisk gives us the value of the array element whose address is currently stored in numptr. In the for loop in Example 17, the value that numptr points to is printed and then numptr is incremented so that it points to the next array element.

The overall thing to keep in mind is that in order to determine what you will get from a certain pointer expression requires that you know the type of pointer and what it points to.

If num is a pointer to an int, then *num gives you the value of the int that num points to. If num is a two-dimensional array, then *num will give you the address of the beginning of the array.

References