Ch 9. Pointers

A Quote

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the goto statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.


9.1 The Address Operator

- Consider main memory to be a sequence of consecutive cells (1 byte per cell).
- The cells are numbered (like an array). The number of a cell is its address.
- When your program is compiled, each variable is allocated a sequence of cells, large enough to hold a value of its type.
- The address operator (&) returns the address of a variable.

```
int x = 99;
cout << x << endl;
cout << &x << endl;
```

Output:

```
99
0xbffffb0c
```

- Addresses in C/C++ are displayed in hexadecimal. [bffffb0c = 3,221,224,204]

9.2 Pointer Variables

- A pointer variable (or pointer):
  - contains the address of a memory cell
- An asterisk is used to define a pointer variable
  ```
  int *ptr;
  ```
- “ptr is a pointer to an int” or
- “ptr can hold the address of an int”
  ```
  int * ptr; //same as above
  int* ptr; //same as above
  ```
Using Pointer Variables

• Assigning an address to a pointer variable:

```cpp
int x = 99;
int *ptr;
ptr = &x;
cout << x << endl;
cout << ptr << endl;
```

Output:

```
address of x: 0xbffffb0c
```

Another example

• Assigning an address to a pointer variable:

```cpp
int rate = 100;
int *s_rate;
s_rate = &rate;
cout << rate << endl;
cout << s_rate << endl;
```

Output:

```
s_rate
```

Dereferencing Operator: *

• The unary operator * is the indirection or dereferencing operator.
• It allows you to access the item that the pointer points to.
• *ptr is an alias for the variable that ptr points to.

```cpp
int x = 1;
int y = 2;
int *ip;
ip = &x; // ip points to x
y = *ip; // y is assigned what ip points to
*ip = 100; // (the variable ip points to) gets 100
```

pointer declaration vs. dereferencing

• The asterisk is used in 2 different contexts for pointers, meaning two different things

1. To declare a pointer, in a variable definition:

   ```cpp
   int *ip; // ip is defined to be a pointer to an int
   ```

2. To dereference a pointer, in an expression

   ```cpp
   y = *ip; // y is assigned what ip points to
   ```
Dereferencing Operator

- Another example

```c
int x = 25, y = 50, z = 75;
int *ptr;
ptr = &x;
*ptr = *ptr + 100;
ptr = &y;
*ptr = *ptr + 100;
ptr = &z;
*ptr = *ptr + 100;
```

```
cout << x << " " << y << " " << z << endl;
```

9.3 Pointers and Arrays

- You can treat an array variable as if it were a pointer to its first element.

```c
int numbers[] = {10, 20, 30, 40, 50};
```

```
cout << "first: " << numbers[0] << endl;
cout << "first: " << *numbers << endl;
cout << &numbers[0] << endl;
cout << numbers << endl;
```

Output:
```
first: 10
first: 10
0xbffffb00
0xbffffb00
```

Pointer Arithmetic

- When you add a value to a pointer, you are actually adding that value times the size of the data type being referenced by the pointer.

```
int numbers[] = {10, 20, 30, 40, 50};
// sizeof(int) is 4.
// Let us assume numbers is stored at 0xbffffb00
// Then numbers+1 is really 0xbffffb00 + 1*4, or 0xbffffb04
// And numbers+2 is really 0xbffffb00 + 2*4, or 0xbffffb08
// And numbers+3 is really 0xbffffb00 + 3*4, or 0xbffffb0c
```

```
cout << \"second: \" << numbers[1] << endl;
cout << \"second: \" << *(numbers+1) << endl;
cout << \"size: \" << sizeof(int) << endl;
cout << numbers << endl;
cout << numbers+1 << endl;
```

Output:
```
second: 20
second: 20
size: 4
0xbffffb00
0xbffffb04
```

- Note: array[index] is equivalent to *(array + index)
Pointers and Arrays

- pointer operations can be used with array variables.
  ```c
  int array[10];
  cin >> *(array+3);
  ```
- subscript operations can be used with pointers.
  ```c
  int array[] = {1,2,3};
  int *ptr = array;
  cout << ptr[2];
  ```
- You cannot change the value of the array variable.

```c
double totals[20];
double *dptr;
dptr = totals; //ok
totals = dptr; //not ok, totals is a const
```

9.4 Pointer Arithmetic

- Operations on pointers over data type d:
  ```c
  d *ptr;
  ```
  - ptr+n where n is int: ptr+n*sizeof(d)
  - ptr–n where n is int: ptr–n*sizeof(d)
  - ++ and --: ptr=ptr+1 and ptr=ptr-1
  - += and -=
  - subtraction: ptr1 – ptr2
    result is number of values of type d between the two pointers.

9.5 Initializing Pointers

- Pointers can be initialized as they are defined.
  ```c
  int myValue;
  int *pint = &myValue;
  ```
  ```c
  int ages[20];
  int *pint1 = ages;
  ```
  ```c
  int *p1 = &myValue, *p2=ages, x=1;
  ```
  ```c
  double x, y, *d, radius;
  ```

- Note: pointers to data type d can be defined along with other variables of type d.

- int arr[25];
  ```c
  cout << arr[1] > &arr[0] << endl;
  cout << arr == &arr[0] << endl;
  cout << arr <= &arr[20] << endl;
  cout << arr > arr+5 << endl;
  ```
- What is the difference?
  ```c
  - ptr1 < ptr2
  - *ptr1 < *ptr2
  ```
9.7 Pointers as Function Parameters

- Use pointers to implement pass by reference.

```c
//prototype: void changeVal(int *);
void changeVal (int *val) {
    *val = *val * 11;
}
int main() {
    int x;
    cout << "Enter an int " << endl;
    cin >> x;
    changeVal(&x);
    cout << x << endl;
}
```

- How is it different from using reference parameters?

9.8 Dynamic Memory Allocation

- When a function is called, memory for local variables is automatically allocated.
- When function exits, memory for local variables automatically disappears.
- Must know ahead of time the maximum number of variables you may need.
- Dynamic Memory allocation allows your program to create variables on demand, during run-time.

```c
int *iptr;
iptr = new int;
*iptr = 11;
 cin >> *iptr;
int value = *iptr / 3;
```

Pointers as array parameter

- Pointer may be used as a parameter for array

```c
double totalSales(double *arr, int size) {
    double sum = 0.0;
    for (int i=0; i<size; i++) {
        sum += arr[i];               //OR: sum += *arr++;
    }
    return sum;
}
int main() {
    double sales[4];
    // input data into sales here
    cout << "Total sales: " << totalSales(sales, 4) << endl;
}
```

- What?

```c
sum += *arr++;
sum = sum + *arr;
arr = arr+1;
```

The new operator

- “new” operator requests dynamically allocated memory for a certain data type:

```
int *iptr;
iptr = new int;
```

- new operator returns address of newly created anonymous variable.
- use dereferencing operator to access it:

```
*iptr = 11;
cin >> *iptr;
int value = *iptr / 3;
```
Dynamically allocated arrays

- dynamically allocate arrays with new:

```cpp
int *iptr; // for dynamically allocated array
int size;
cout << "Enter number of ints: ";
cin >> size;
iptr = new int[size];
for (int i=1; i<size; i++) {
    iptr[i] = i;
}
```

Make sure new succeeded

- new will fail if not enough memory available.
- new returns NULL (which is 0) if it fails.
- A pointer whose value is 0 is called a “Null pointer”.

```cpp
iptr = new int[10000];
if (iptr == NULL) {
    cout << "Error allocating memory." << endl;
    return EXIT_FAILURE;
}
```

delete!

- When you are finished using a variable created with new, use the delete operator to destroy it:

```cpp
int *ptr;
double *array;
ptr = new int;
array = new double[25];
...
delete ptr;
delete [] array; // note [] required for dynamic arrays!
```

- Do not “delete” pointers whose values were NOT dynamically allocated using new!
- Do not forget to delete dynamically allocated variables (Memory Leaks!!).

9.9 Returning Pointers from Functions

- functions may return pointers:

```cpp
int *findZero (int arr[]) {
    int *ptr;
    ptr = arr;
    while (*ptr != 0)
        ptr ++;
    return ptr;
}
```

- The returned pointer must point to
  - dynamically allocated memory OR
  - an item passed in via an argument

NOTE: the return type of this function is (int *) or pointer to an int.
Returning Pointers from Functions: \texttt{duplicateArray}

\begin{verbatim}
int *duplicateArray (int *arr, int size) {
    int *newArray;
    if (size <= 0) //size must be positive
        return NULL;
    newArray = new int [size]; //allocate new array
    for (int index = 0; index < size; index++)
        newArray[index] = arr[index]; //copy to new array
    return newArray;
}
\end{verbatim}

\begin{tabular}{|c|c|c|}
\hline
int a [5] = {11, 22, 33, 44, 55}; & 0 ok & 25 \\
int *b = duplicateArray(a, 5); & 1 ok & \\
for (int i=0; i<5; i++)
    if (a[i] == b[i])
        cout << i << " ok" << endl;
delete [] b; & 2 ok & \\
& 3 ok & \\
& 4 ok & \\
\hline
\end{tabular}

Problems returning pointers (watchout)

- \textbf{Bad:} \begin{verbatim}
int *getList() {
    int list[80];
    for (int i = 1; i<80; i++)
        list[i] = i;
    return list;
}
\end{verbatim}
- \textbf{Good:} \begin{verbatim}
int *getList () {
    int *list;
    list = new int[80];
    for (int i=1; i<80; i++)
        list[i] = i;
    return list;
}
\end{verbatim}

Memory Leak!

\begin{verbatim}
int *appendArray (int x, int *arr, int size)
{
    int *newArray;
    newArray = new int [size+1]; //allocate new array
    for (int index = 0; index < size; index++)
        newArray[index] = arr[index]; //copy to new array
    newArray[size] = x; //add x to last spot
    return newArray;
}
\end{verbatim}

// in main: set up list with one element
int *myList;
int size = 1;
myList = new int [1];
myList[0] = 11;
//cont. ...
Memory Leak: solved

// this code is performed often
int z;
cout << "Enter int for z: ";
cin >> z;

int *newList; //temp variable, local to the case block
newList = appendArray(z, myList, size);
delete [] myList;
myList = newList;

This cell was deleted/released before re-assigning the pointer.