Pointers &
Dynamic Memory Allocation

Unit 3
Chapter 9
CS 2308
Spring 2019
Jill Seaman

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.

A Quote
From: “The C Programming Language (2nd ed.),” Brian W. Kernighan and

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.

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

Output:
```
99
0xbffffb0c
```

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
  ```c
  int *ptr;
  ```
• “ptr is a pointer to an int” or
• “ptr can hold the address of an int”

```c
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;
```

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:

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

In this example assume an int and an address require only 1 byte of memory for storage.

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) is assigned 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 use an array variable (the name of the array) as if it were a pointer to its first element.

```c
int numbers[] = {10, 20, 30, 40, 50};
cout << "first: " << numbers[0] << endl;
cout << "second: " << *(numbers+1) << endl;
cout << "size: " << sizeof(int) << endl;
```

Pointers and Arrays

- 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.

```c
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
```

Pointer and Arrays

- Note unary * has higher precedence than +, so the parentheses are required.

```c
int numbers[] = {10, 20, 30, 40, 50};
cout << "second: " << *(numbers+1) << endl;
cout << "size: " << sizeof(int) << endl;
cout << *(numbers+1) << endl;
```

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

- Pointer operations can be used with array variables.
  ```cpp
  int list[10];
  cin >> *(list+3);
  ```

- Subscript operations can be used with pointers.
  ```cpp
  int list[] = {1,2,3};
  int *ptr;
  ptr = list;
  cout << ptr[2];
  ```

- Only difference: you cannot change the value of the array variable.
  ```cpp
  double totals[20];
  double *dptr;
  dptr = totals; // ok
  totals = dptr; // wrong!!, totals is a const
  ```

9.5 Initializing Pointers

- Pointers can be initialized when they are defined.
  ```cpp
  int myValue;
  int *pint = &myValue;
  ```

Note: you are initializing the pointer, NOT the variable the pointer points to.

- Note: pointers to data type d can be defined along with other variables of type d.
  ```cpp
  double x, y, *d, radius;
  ```

9.7 Pointers as Function Parameters

- Use pointers to implement pass by reference.
  ```cpp
  // 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 the syntax different from using reference parameters?

Pointers as array parameter

- Pointer may be used as a parameter for array arg
  ```cpp
  double totalSales(double *arr, int size) {
    double sum = 0.0;
    for (int i=0; i<size; i++) {
      sum += arr[i]; // or sum += *(arr+i);
    }
    return sum;
  }

  int main() {
    double sales[4];
    // input data into sales here
    cout << "Total sales: " << totalSales(sales, 4) << endl;
  }
  ```
9.4 Pointer Arithmetic

- Operations on pointers to data type `d`:
  
  ```
  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`
    changes `ptr` to point to next/prev variable of type `d`
  - `+=` and `-=`
  - subtraction: `ptr1 – ptr2`
    result is number of values of type `d` between the two pointers.

9.6 Comparing Pointers

- Pointers may be compared using relational operators (based on their address values):
  `<  <=  >  >=  ==  !=`

- Examples:
  ```
  int arr[25];
  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?
  - `ptr1 < ptr2`
  - `*ptr1 < *ptr2`

9.8 Dynamic Memory Allocation

- When a function is called, memory for local variables is automatically allocated.
- When a 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.

The new operator

- “new” operator requests dynamically allocated memory for a certain data type:
  ```
  int *iptr;
  iptr = new int;
  ```
- New operator returns the address of a 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
type *iptr;  //for dynamically allocated array
int size;

cout << "Enter number of ints to be stored: ";
cin >> size;
iptr = new int[size];

for (int i=0; i<size; i++) {
    iptr[i] = i;         // populating the array
}
```

- Program will throw an exception and terminate if not enough memory available to allocate

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.

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

Returning Pointers from Functions: duplicateArray

```cpp
int *duplicateArray (int *arr, int size) {
    int *newArray;
    if (size <= 0)         //size must be positive
        return NULL;        //NULL is 0, an invalid address
    newArray = new int [size];  //allocate new array
    for (int index = 0; index < size; index++)
        newArray[index] = arr[index];  //copy to new array
    return newArray;
}
```

- 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!!).

Output

<table>
<thead>
<tr>
<th>Output</th>
<th>0 ok</th>
<th>1 ok</th>
<th>2 ok</th>
<th>3 ok</th>
<th>4 ok</th>
</tr>
</thead>
</table>
Problems returning pointers
(watchout)

- **Bad:**
  ```
  int *getList() {
    int list[80];
    for (int i = 0; i<80; i++)
      list[i] = i;
    return list;
  }
  ```
  - what happens to list on function exit?

- **Good:**
  ```
  int *getList () {  
                    int *list;
                    list = new int[80];  
                    for (int i=1; i<80; i++)
                      list[i] = i;
                    return list;
  }
  ```

Variable Length Arrays

- Some compilers allow you to use a variable to define the size of a regular (static) array:
  ```
  void f() {
    int size;
    cout << “Enter list length:” << endl;
    cin >> size;
    string list[size]; //size determined at runtime
    ... }
  ```
  - what happens to list on function exit?

- Like dynamic arrays, size is determined at runtime
- Unlike dynamic arrays, array is deleted/deallocated at the end of the function.
- This is NOT a feature of standard C++!!