C and Linux Programming
Eastern Washington University
Computer Science
March 30th – June 12th, 2020

Lecture 5

C Pointers and Dynamic Memory

Joe Dumoulin

Mar 30, 2020

Eastern Washington University

C Pointers and Dynamic Memory

The Story So Far

Basic C syntax

- Basic C Syntax
- Function Basics
- File Handling

Next we will extend these ideas to understand pointers and dynamic memory management in C and Linux programming.

A Closer Look at C Variables

We use variables to store values while our program is running. But what is happening under the covers?

One way to think about variables is as a reserved area of the computer's memory to store a value.

Unlike files, variables only exist while a program is running.

A variable has four aspects:

- A **type**,
- A **name**,
- A **value** (or possibly it may be uninitialized)
- An **address** in memory.

Definition of a pointer

A pointer is a variable whose value is the memory address of another variable.

Consider the following code:

```c
int main () {

  int  var1;
  char var2[10];

  printf("Address of var1 variable: %p\n",  &var1  );
  printf("Address of var2 variable: %p\n",  &var2  );

  printf("var2 points to the beginning of a string. \
      \nIt also has a pointer value:%p\n", var2);

  return 0;
}
```

This code will print out the addresses assigned to these different variables. You can find the address of any variable by prefixing the variable name with "&".

Pointers and Addresses

An example of the output from the code above on a computer with 64-bit addressing is:

```
Address of var1 variable:      0x7ffc56c156e8
Address of var2 variable:      0x7ffc56c156ee
var2 points to the beginning of a string.
It also has a pointer value:   0x7ffc56c156ee
```

Note that these addresses will be different each time the program runs!

Pointers as Variables

Here are some examples of pointer variables in C:

```
int x = 0;     // an integer variable initialized to 0;
int* px = x;   // a pointer-to-integer variable
               //   initialized to the address of x;

double pt1[3] = {5,7,6}; // a point in 3-D.
double* ppt1 = pt1;      // a pointer to the first element.
ppt1 += 1;               // now points to the second element.


// An array of strings (a 2-D array of characters)
char* strings[] =
        {"An old take", "on a modern problem."};
```

C does not have reference variables. Instead it uses pointer variables.

Pointer variables always include a type.

Consider the problem of swapping a value between two integer variables. we can do this with pointers.

```
void swapints(int* p1, int* p2)
{
  // put the contents of memory location p1
  // in a local variable.
  int swp = *p1;

  // swap the contents of address p2 into
  // address p1
  *p1 = *p2;

  // copy local variable value to address p2.
  *p2 = swp;
}
```

Using swapints

The following main() function demonstrates how to use swapints:

```c
int main(void)
{
  int x1 = 50;
  int x2 = 63;
  printf("before swap, x1 = %d, x2 = %d.\n",
      x1, x2);

  swapints(&x1, &x2);

  printf("after swap, x1 = %d, x2 = %d.\n",
      x1,x2);
}
```

A practical example of this would be sorting an array of integers.

Pointers to Arrays

A C array is really nothing more than a pointer to the first element of the array. There is one crucial difference though. The array variable's value is an address, but it is **constant**. It cannot change.

Sometimes array variables are refered to as constant pointers. We will have more to say about constant variables later.

we can get a non-constant pointer to the beginning of an array by assigning a pointer the value of the array variable like this:

```
// the value of arr is the address of the first element.
int arr[] = {1,2}; // arr is constant.

// the value of p is also the address of the first element of ar
int* p = arr;
```

Example: Copying Strings

Since strings are arrays in C, we can use them to illustrate some of the ways you can work with pointers in arrays. For example, consider copying a string into a buffer. The code might look like the below function;

```
#define BUF_SZ 200

int main(void)
{
  char* str = "Be like a headland of rock on \
which the waves break incessantly but it \
stands fast and around it the seething \
of the waters sinks to rest.";

  char buffer[BUF_SZ] = "";
  int i;

  // copy str to buffer, then print buffer
  for (i=0; i<strlen(str); i++) {
    buffer[i] = str[i];
  }
  printf("%s\n", buffer);

  return EXIT_SUCCESS;
}
```

Example: Copying Strings

The above works, but C offers some alternatives that involve pointers. Below is a function for the copy:

```
// copy the contents of p1 to p2.
// Assume '\0' ending strings.
void stringcpy(char* p1, char* p2)
{
  while(*p1) {  // while p1 != 0 (note: 0 == '\0')
    *p2 = *p1;
    p1++;
    p2++;
  }
}
  // in main we call stringcpy like this:
  stringcpy(str, buffer);
  ...
```

Example: Copying Strings

We can use a principle of C programming to make stringcpy even more compact. First note that the assignment operator returns the value of the assignment. we can then move the assignment into the while predicate:

```
void stringcpy(char* p1, char* p2)
{
  while(*p2 = *p1) { // while *p1 != 0,
    p1++;
    p2++;
  }
}
```

Example: Copying Strings

Next we use the precedence of operators to further condense this expression. Note that ++ on the right is applied to the variable *after* *. that means we can get the value from *p1 and assign it to *p2 before we increment p1 and p2 with this line:

```
*p2++ = *p1++;
```

With that, we can change the function to:

```
void stringcpy(char* p1, char* p2)
{
  while(*p2++ = *p1++)  // while *p1 != 0,
    ;
}
```

Pointers and Dynamic Memory

Most languages have the ability to create new objects. The basic C language has no ability to do this. But we can use the C library call **malloc** to help us accomplish the same thing.

Creating objects has two steps.

- Reserve meomory for the object.
- Initialize the obect.

Most languages combine these steps into a single operation called "new". In C, we have to do each of these separately.

Managing Dynamic Memory

Most languages provide a means to clean up memory. Usually they use a system called a **Garbage Collector**. A Garbage Collector (GC) is part of a language system and it keeps track of all allocated objects until they are no loger referenced. Once objects are no longer referenced, the GC removes them from memory.

C has no GC. You must manually remove objects that were allocated with **malloc**. The C library provides the function **free** to accomplish this.

Managing Dynamic Memory - An Example

Here is an example program that allocates a string instead of using a static buffer like we did before.

```
int main(void)
{
  char* quote = "Be like a headland of rock on \
which the waves break incessantly but it \
stands fast and around it the seething \
of the waters sinks to rest.";
  printf("original string:\n\n%s\n\n", quote);

  char* copyquote; // a pointer to nowhere... yet.

  // malloc gives us a memory location for our
  // copy of the quote.
  copyquote = (char*) malloc(strlen(quote));

  // now copy the quote into the new space
  stringcpy(quote, copyquote);

  printf("copied string:\n\n%s\n\n", copyquote);

  // free the allocated memory when we are done.
  free(copyquote);
}
```

A Closer Look at malloc

malloc has some important new aspects. here's the code again:

```
...
  copyquote = (char*) malloc(strlen(quote));
...
```

- **copyquote** is an uninitialized pointer variable.
- **(char*)** is a **cast** from the return value from **malloc**. More about this below.
- **strlen(quote)** gets the proper length of the string (including the terminator '\0'). This is the number of bytes to allocate. **malloc** always takes the number of **bytes** as a parameter.

malloc return type: void *

malloc uses a special type of pointer for its return value: **void***. If **malloc** is successful and allocates the memory asked for, then the pointer returned points to the new memory.

The pointer has type **void *** which must be cast to the type that you want. We will see other examples of **malloc** in the next lecture.

The Next Assignment

In the next assignment, you will change the static buffer from a file copying program to allocate memory dynamically., and then free the buffer memory back to the heap.