Alexandar Devic, Nolan Betts, Raveeja Deshpande, Kevin Brennan

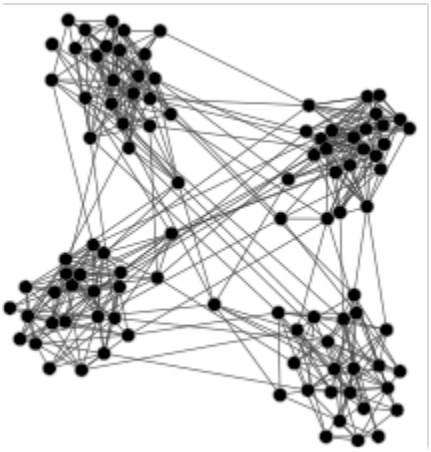
Lecture 4
Dr. Urgaonkar, CMPSCI 311

Pointers

The Fundamentals of Pointers in regards to C Programming

Introduction

Pointers are a useful tool for any aspect of programming, especially when used in C. Pointers are really helpful and let us do some interesting actions when combined with simple arithmetic.



What you'll learn…

What are pointers?	Learn the theory of pointers, its background, main uses, and view some visuals and explanations
Why use pointers?	See what pointers are used for and how they are powerful objects with the operating system.
Using Pointers	Learn how to use pointers through code and see examples of its uses.
Pointer Arithmetic	Learn what pointer arithmetic is and how it can be a powerful tool.

What are Pointers?

As you can probably tell, pointers 'point' to something. A pointer is a variable that points to another variable by holding its virtual address (see *Memory Management* for more information on virtual addresses).

[From the scribes, not shown in lecture]

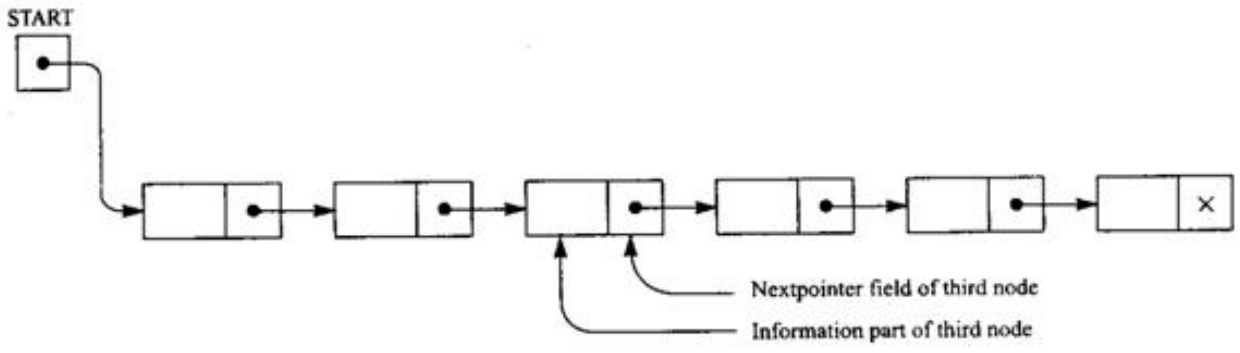
Let's think of an example. In astronomy, many stars and planets are described by their angles in the relative sky. For example, the North Star, is described as 0^0 North, 40^0 above the horizon. Suppose the following list is given to you:

North Direction	Horizon Direction	Object
0^0	40^0	North Star
10^0	40^0	???
30^0	10^0	???

How would you go about finding the unknown objects? You could use a telescope! Just like how a pointer stores an address space, a telescope acts the same. When you zero in the telescope, with both the North and Horizontal directions, your telescope is *pointing* to some object. And, just like in C when you dereference a pointer, you do so in a telescope by looking through the eye-lens. Only then can you see what is there. Pointers work one in the same way. Once you store elements, say integers into memory, they are given virtual addresses, and these can be saved in pointers for use later.

Why use Pointers?

With the power of present day computers, the purpose of pointers can often be a murky area. As we know, 'C' was a language that was created in the 1970's when computers had far less capabilities. In 'C' there is no pass by reference. Pointers can come in handy when you want to complete a similar task to pass by reference. Pointers can also be useful in 'C' for traversing arrays or strings quickly. Pointers are also useful for constructing data structures such as stacks, queues, lists or search trees.



Using Pointers in C.

Pointers have types. You can declare pointers as any other primitive, including void. Many times, you will know the object you are pointing to, and so you should instantiate your pointers with that type, but if in the event you do not, using the 'void' type acts like a *wildcard* type, it can be whatever you want it to be, and you can type cast this pointer whenever needed. A table below illustrates some possible pointer declarations:

C Code	Explanation
`int *my_pointer;`	Declares a pointer for integers.
`bool *my_pointer;`	Declares a pointer for booleans.
`double *my_pointer;`	Declares a pointer for doubles.
`float *my_pointer;`	Declares a pointer for floating numbers.
`void *my_pointer;`	Declares a pointer for any type.

Some things to notice is the ***** character before our variable names. This is what's called a *dereference token*. Using pointers, there are two tokens to keep in mind:

C Token	Meaning
*	"Dereference" – gets the ***value*** at the address at the pointed object
&	"Address Of" – gets the ***address*** (hexadecimal) of the pointed object

Now knowing this, an example of a pointer implementation is shown below:

```
int i; //integer i

i = 100; //i holds value 100

int*p; //pointer to an integer

p = &I; //p points to i
```

Pointers are valuable for accessing the information of the variable that it points to. If we were to simply print out the pointer based on our implementation we would be printing out the virtual address of the variable.

```
printf("%p", p); //returns virtual address of i
```

However, if we deference the pointer during the print we would see the value contained by the variable.

```
printf("%p", *p); //returns 100
```

Pointer Arithmetic

Remembering that pointers are virtual addresses, what would happen if you went to the next address over, or rather, n times before or after? If we're not careful, pointers can miss-point us to other non-desired addresses. However, if we keep track, and understand what memory addresses are physically (a hexadecimal number corresponding to a memory location) we can do some very impressive things. This is what pointer arithmetic is all about.

Let's say we have some pointer **p**, from the previous example, and add one to itself. What do you think will happen? To understand this, we need to look at the address of **p**. Let's say, for simplicity sake, your computer contains addresses 1-1000. When we declare an integer, it is assigned the next available address. Integer **i** in our case happens to occupy address 85. 85 is what is stored in the pointer variable **p**, as it is the *address of where integer **i** resides*. So, if we were to say **p** += 1, then we are saying that instead of pointing at memory location 85, we want now to be looking at memory location 86. This is pointer arithmetic.

An interesting consequence of this is we can instantiate our own arrays with variable size. In most languages, arrays *must* be declared with some constant value *before* runtime. But with pointers and C, you can get around this.

```
int *p; // integer pointer

//refer to the malloc pages for this implementation.

p = malloc ( (n_size) * sizeof( int ) );

//p is now an array of variable size n_size!!
```

Now with p being an array of variable length, let's manipulate this to obtain and set values using pointer arithmetic.

Pointer Arithmetic	Array Equivalent
*p	array[0]
*(p+0)	array[0]
*(p+1)	array[1]
*(p+5)	array[5]
*(p+n)	array[n]

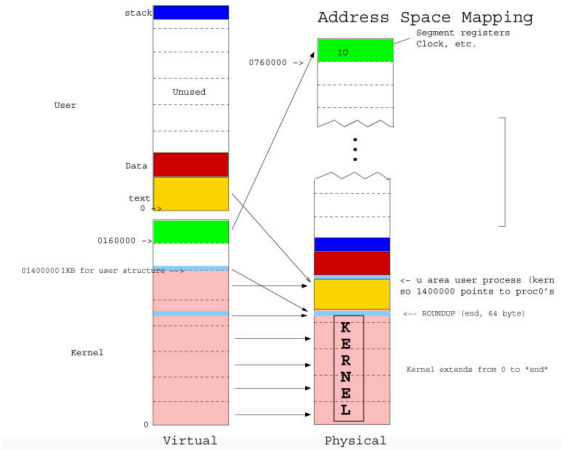
As you can see, we can allot arrays and other useful data structures without knowing the size of object before runtime.

Memory Management

The Basics of Memory Management in regards to C Programming

Introduction

Memory Management a system level process in which data is stored and read from. A complex process abstracts the details of physical addresses into virtual memory.



What you'll learn…

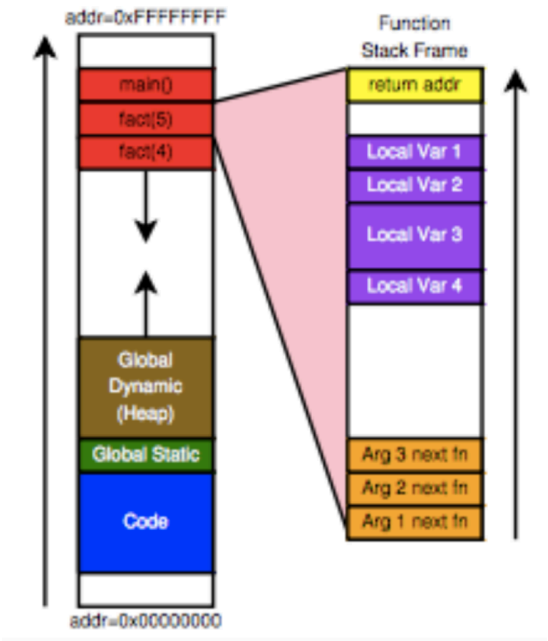
What is Memory Management?	A way to dynamically allocate portions of memory to programs at their request.
Why do we have Memory Management?	To separate programs from one another, and give each a virtual address space that they alone can use.
Using Memory Management	We will learn how the OS utilizes techniques to provide an efficient memory system.
Memory Address Space	We will see how the address space is broken up and utilized.

What is Virtual Memory?

Virtual memory is an illusion of physical memory that is run by the operating system. It is not the same as physical memory, but rather an abstraction that gives each individual process the notion that it is the only one using physical memory. These virtual addresses are then mapped to actual addresses on physical memory.

What is the Address Space?

The address space is the amount of memory allocated for all possible addresses for a computational entity. The address space may refer to a range of either physical or virtual addresses accessible to a processor or reserved for a process.



Since our modern machines are generally 64-bits, the address space on a modern device ranges from 0 to 2^{64}-1. Excluding the Operation System, this space is made up of four parts; the code segment, data segment, stack, and heap. As you should know by now, the stack grows down and the heap goes up when speaking about their addresses.

What is the Purpose of Virtual Memory?

The main purpose of Virtual Memory is isolation. In the beginning of the computer era, the physical memory was used by every process at the same time. This could lead to some serious trouble. When a machine only uses physical addresses, data can easily be corrupted and over-written as multiple processes are utilizing similar spaces in memory. With Virtual Memory each process is isolated and cannot be affected by another program running on that machine.

When physical memory is full an area on the disk is used as memory. This is called the "swap space". The swap space is an area on a hard disk used as the virtual memory extension of a computer's real memory (RAM). Having a swap file allows your computer's operating system to pretend that you have more memory than you actually do.