

CS61CL : Machine Structures

Lecture #2 - C Pointers and Arrays

2009-06-24



Jeremy Huddleston

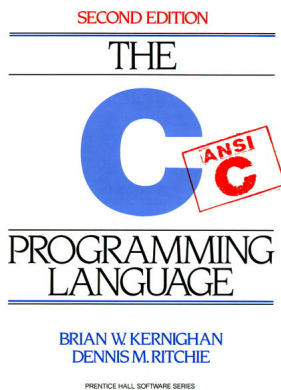


Administrivia

- **Buggy Start**
 - Lab schedule, lab machines, ...
- **HW0 due tomorrow in lab**
- **HW1 due Monday by 9:30am**
 - Assignment online, submission details to come
- **Note Taking Service**
- **Slide Handouts**
- **Room Change (hopefully)**
- **Final Exam: 8/13 9:30-12:30 (3hrs)**



Introduction to C



Has there been an update to ANSI C?

- **Yes! It's called the "C99" standard**
 - You need "gcc -std=c99" to compile
- **References**
 - <http://en.wikipedia.org/wiki/C99>
 - http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html
- **Highlights**
 - Declarations anywhere, like Java (#15)
 - Java-like // comments (to end of line) (#10)
 - Variable-length non-global arrays (#33)
 - <inttypes.h>: explicit integer types (#38)
 - <stdbool.h> for boolean logic defs (#35)
 - restrict and inline keywords for optimization (#30-32)



Compilation : Overview

- C **compilers** turn C code into **architecture specific** assembly code. An **assembler** turns this into machine code (string of 1s and 0s).
 - GCC does both
 - Unlike Java which converts to **architecture independent** bytecode.
 - Unlike most Scheme, Python, Ruby environments which interpret the code.
 - These differ mainly in **when** your program is converted to machine instructions.



C Syntax: main

- **To get the main function to accept arguments, use this:**

```
int main (int argc, char *argv[])
```
- **What does this mean?**
 - **argc** will contain the number of strings on the command line (the executable counts as one, plus one for each argument). Here **argc** is 2:

```
$ sort myFile
```
 - **argv** is a pointer to an array containing the arguments as strings (more on pointers later).



C Syntax: Variable Declarations

- Very similar to Java, but with an important difference

- A variable may be initialized in its declaration; **if not, it holds garbage!**

- Examples of declarations:

- correct:

```
int a = 0, b = 10;
...

```

- Incorrect:

```
int i;
while(i < 10) {
...
i++;
}

```



Address vs. Value

- Consider memory to be a single huge array:

- Each cell of the array has an address associated with it.
- Each cell also stores some value.

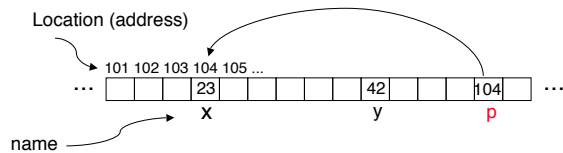
- Don't confuse the **address** referring to a memory location with the **value** stored in that location.



Pointers

- An address refers to a particular memory location. In other words, it points to a memory location.

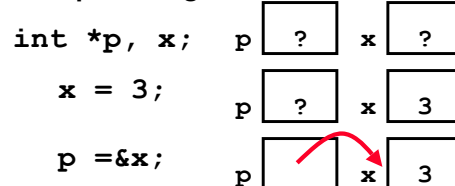
- **Pointer**: A variable that contains the address of a variable.



Pointers

- How to create a pointer:

& operator: get address of a variable



Note the "&" gets used 2 different ways in this example. In the declaration to indicate that **p** is going to be a pointer, and in the **printf** to get the value pointed to by **p**.

- How get a value pointed to?

* "dereference operator": get value pointed to

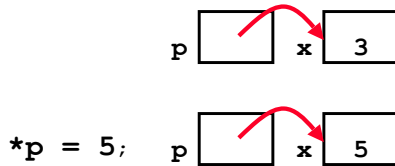
```
printf("p points to %d\n", *p);
```



Pointers

- How to change a variable pointed to?

- Use dereference * operator on left of =



Pointers and Parameter Passing (1/4)

- C passes parameters "by value"

- procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original

- What gets printed?

```
void AddOne(int x)
{ x = x + 1; }
```

y = 5

```
int y = 5;
AddOne(y);
printf("y = %d\n", y);
```



Pointers and Parameter Passing (2/4)

- Solved by passing in a **pointer** to our subroutine.
- Now what gets printed?

```
void AddOne(int *p)           y = 6
{   *p = *p + 1;   }

int y = 5;
AddOne(&y);
printf("y = %d\n", y);
```



CS61CL L01 Introduction (13)

Huddleston, Summer 2009 © UC

Pointers and Parameter Passing (3/4)

- But what if what you want changed is a **pointer**?
- What gets printed?

```
void IncrementPtr(int *p)     *q = 50
{   p = p + 1;   }

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
printf("*q = %d\n", *q);
```



CS61CL L01 Introduction (14)

Huddleston, Summer 2009 © UC

Pointers and Parameter Passing (4/4)

- Solution! Pass a **pointer to a pointer**, declared as ****h**
- Now what gets printed?

```
void IncrementPtr(int **h)    *q = 60
{   *h = *h + 1;   }

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(&q);
printf("*q = %d\n", *q);
```



CS61CL L01 Introduction (15)

Huddleston, Summer 2009 © UC

Pointers

- Pointers are used to point to **any** data type (int, char, a struct, etc.).
- Normally a pointer can only point to one type (int, char, a struct, etc.).
 - void * is a type that can point to anything (generic pointer)
 - Use sparingly to help avoid program bugs... and security issues... and a lot of other bad things!



CS61CL L01 Introduction (16)

Huddleston, Summer 2009 © UC

C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- Local variables in C are not initialized, they may contain anything.
- What does the following code do?

```
void f()
{
    int *ptr;
    *ptr = 5;
}
```



CS61CL L01 Introduction (17)

Huddleston, Summer 2009 © UC

Arrays (1/5)

- **Declaration:**

```
int ar[2];
```

declares a 2-element integer array. *An array is really just a chunk of memory.*

```
int ar[] = {795, 635};
```

declares and fills a 2-elt integer array.

- **Accessing elements:**

```
ar[num]
```



returns the numth element.

CS61CL L01 Introduction (18)

Huddleston, Summer 2009 © UC

Arrays (2/5)

- Arrays are (almost) identical to pointers
 - `char *string` and `char string[]` are nearly identical declarations
 - They differ in very subtle ways: incrementing, declaration of filled arrays
- **Key Concept:** An array variable is a “pointer” to the first element.



Arrays (3/5)

- Consequences:
 - `ar` is an array variable but looks like a pointer in many respects (though not all)
 - `ar[0]` is the same as `*ar`
 - `ar[2]` is the same as `*(ar+2)`
 - We can use pointer arithmetic to access arrays more conveniently.
- Declared arrays are only allocated while the scope is valid

```
char *foo() {
    char string[32]; ...;
    return string;
} is incorrect
```



Arrays (4/5)

- Array size `n`; want to access from 0 to `n-1`, so you should use counter AND utilize a constant for declaration & incr
 - Wrong

```
int i, ar[10];
for(i = 0; i < 10; i++){ ... }
```
 - Right

```
#define ARRAY_SIZE 10
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```
- Why? **SINGLE SOURCE OF TRUTH**
 - You re utilizing **indirection** and **avoiding maintaining two copies** of the number 10



Arrays (5/5)

- Pitfall: An array in C does **not** know its own length, & bounds not checked!
 - Consequence: We can accidentally access off the end of an array.
 - Consequence: We must pass the array **and its size** to a procedure which is going to traverse it.
- **Segmentation faults and bus errors:**
 - These are VERY difficult to find; be careful! (You ll learn how to debug these in lab...)



Pointer Arithmetic (1/2)

- Since a pointer is just a mem address, we can add to it to traverse an array.
- `p+1` returns a ptr to the next array elt.
- ***p++ vs (*p)++ ?**
 - `x = *p++` \Rightarrow `x = *p ; p = p + 1 ;`
 - `x = (*p)++` \Rightarrow `x = *p ; *p = *p + 1 ;`
- What if we have an array of large structs (objects)?
 - C takes care of it: In reality, `p+1` doesn't add 1 to the memory address, it adds the **size of the array element**.



Pointer Arithmetic (2/2)

- C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.
 - 1 byte for a char, 4 bytes for an int, etc.
- So the following are equivalent:

```
int get(int array[], int n)
{
    return (array[n]);
    // OR...
    return *(array + n);
}
```



Pointers in C

- Why use pointers?
 - If we want to pass a huge struct or array, it's easier to pass a pointer than the whole thing.
 - In general, pointers allow cleaner, more compact code.
- So what are the drawbacks?
 - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
 - **Dangling reference** (premature free)
 - **Memory leaks** (tardy free)



C Strings

- A **string** in C is just an array of characters.

```
char string[] = "abc";
```
- How do you tell how long a string is?
 - Last character is followed by a 0 byte (null terminator)

```
int strlen(char s[])  
{  
    int n = 0;  
    while (s[n] != 0) n++;  
    return n;  
}
```



“And in Conclusion...”

- A **pointer** is a C version of the address.
 - * “follows” a pointer to its value
 - & gets the address of a variable
- Pointers and arrays are **virtually same**
- C knows how to **increment pointers**
- C is an efficient language, with little protection
 - Array bounds **not checked**
 - Variables **not** automatically initialized
- (Beware) The cost of efficiency is more overhead for the programmer.
 - “C gives you a lot of extra rope but be careful not to hang yourself with it!”



Reference slides

You ARE responsible for the material on these slides (they're just taken from the reading anyway); we've moved them to the end and off-stage to give more breathing room to lecture!



Corrections Errors from last lecture

- The average course GPA was listed in one place as 2.9 and the other as B- (2.7). The correct average is B- (2.7)
- The midterm will be a 1 hour exam which you have 1.5 hours to take.



Administrivia

- Finish reading K&R by the next lecture
- There is a language called D!
 - www.digitalmars.com/d/
- Homework expectations
 - Readers don't have time to fix your programs which have to run on lab machines.
 - Code that doesn't compile or fails all of the autograder tests ⇒ 0



Administrivia

- Slip days
 - You get 3 “slip days” to use for any homework assignment or project
 - They are used at 1-day increments. Thus 1 minute late = 1 slip day used.
 - They are recorded automatically (by checking submission time) so you don't need to tell us when you are using them
 - Once you've used all of your slip days, when a project/hw is late, it's ... 0 points.
 - If you submit twice, we ALWAYS grade the latter, and deduct slip days appropriately
 - You no longer need to tell anyone how your dog ate your computer.
 - You should really save for a rainy day ... we all get sick and/or have family emergencies!



Pointers & Allocation (1/2)

- After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet (*it actually points somewhere - but don't know where!*). We can either:

- make it point to something that already exists, or
- allocate room in memory for something new that it will point to... (next time)



Pointers & Allocation (2/2)

- Pointing to something that already exists:

```
int *ptr, var1, var2;  
var1 = 5;  
ptr = &var1;  
var2 = *ptr;
```

- `var1` and `var2` have room implicitly allocated for them.



Arrays (one elt past array must be valid)

- Array size `n`; want to access from 0 to `n-1`, but test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;  
...  
p = &ar[0]; q = &ar[10];  
while (p != q)  
    /* sum = sum + *p; p = p + 1; */  
    sum += *p++;
```

- Is this legal?

- C defines that one element past end of array **must be a valid address**, i.e., not cause a bus error or address error



Pointer Arithmetic

- So what's valid pointer arithmetic?
 - Add an integer to a pointer.
 - Subtract 2 pointers (in the same array).
 - Compare pointers (<, <=, ==, !=, >, >=)
 - Compare pointer to `NULL` (indicates that the pointer points to nothing).
- Everything else is illegal since it makes no sense:
 - adding two pointers
 - multiplying pointers
 - subtract pointer from integer



Pointer Arithmetic to Copy memory

- We can use pointer arithmetic to “walk” through memory:

```
void copy(int *from, int *to, int n) {  
    int i;  
    for (i=0; i<n; i++) {  
        *to++ = *from++;  
    }  
}
```

- Note we had to pass size (`n`) to `copy`



Arrays vs. Pointers

- An array name is a read-only pointer to the 0th element of the array.
- An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.

```
int strlen(char s[])    int strlen(char *s)
{
    int n = 0;          {
    while (s[n] != 0)   while (s[n] != 0)
        n++;           n++;
    return n;          return n;
}
```

Could be written:
while (s[n])



CS61CL L01 Introduction (37)

Huddleston, Summer 2009 © UCB

Pointer Arithmetic Summary

- $x = *(p+1) ?$
 $\Rightarrow x = *(p+1) ;$
- $x = *p+1 ?$
 $\Rightarrow x = (*p) + 1 ;$
- $x = (*p)++ ?$
 $\Rightarrow x = *p ; *p = *p + 1 ;$
- $x = *p++ ? (*p++) ? *(p)++ ? *(p++) ?$
 $\Rightarrow x = *p ; p = p + 1 ;$
- $x = ++p ?$
 $\Rightarrow p = p + 1 ; x = *p ;$
- Lesson?



Using anything but the standard $*p++$, $(*p)++$ causes more problems than it solves!

CS61CL L01 Introduction (38)

Huddleston, Summer 2009 © UCB

Segmentation Fault vs Bus Error?

- <http://www.hyperdictionary.com/>

• Bus Error

• A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include **invalid address alignment** (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error. A bus error triggers a processor-level exception which Unix translates into a "SIGBUS" signal which, if not caught, will terminate the current process.

• Segmentation Fault

• An error in which a running Unix program attempts to **access memory not allocated** to it and terminates with a segmentation violation error and usually a core dump.



CS61CL L01 Introduction (39)

Huddleston, Summer 2009 © UCB

More C Pointer Dangers

- Unlike Java, C lets you **cast** a value of any type to any other type without performing any checking.

```
int x = 1000;
int *p = x;           /* invalid */
int *q = (int *) x; /* valid */
```

- The first pointer declaration is invalid since the types do not match.
- The second declaration is valid C but is almost certainly wrong



• Is it ever correct?

CS61CL L01 Introduction (40)

Huddleston, Summer 2009 © UCB

C Strings Headaches

- **One common mistake is to forget to allocate an extra byte for the null terminator.**
- More generally, C requires the programmer to manage memory manually (unlike Java or C++).
 - When creating a long string by concatenating several smaller strings, the programmer must insure there is enough space to store the full string!
 - What if you don't know ahead of time how big your string will be?
 - Buffer overrun security holes!



CS61CL L01 Introduction (41)

Huddleston, Summer 2009 © UCB

Common C Error

- There is a difference between assignment and equality

$a = b$ is assignment

$a == b$ is an equality test

- This is one of the most common errors for beginning C programmers!

• One solution (when comparing with constant) is to put the var on the right!
If you happen to use $=$, it won't compile.

```
if (3 == a) { ...
```



CS61CL L01 Introduction (42)

Huddleston, Summer 2009 © UCB

C String Standard Functions

- `int strlen(char *string);`
 - compute the length of `string`
- `int strcmp(char *str1, char *str2);`
 - return 0 if `str1` and `str2` are identical (how is this different from `str1 == str2?`)
- `char *strcpy(char *dst, char *src);`
 - copy the contents of string `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.

