How about them A’s!! Go Oaktown!!

CS61C - Machine Structures

Lecture 3
C pointers
2002-09-03

⇐ Dan Garcia
(www.cs.berkeley.edu/~ddgarcia)

Dave Patterson ⇒
(www.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/
Review: C syntax reminder

° 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!
Review: Pointer Overview

° 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.
Review: Pointer Usage

Once a pointer is declared:

- use `&` to return a pointer to an existing variable (the memory address of the variable)
- use `*` to return the value pointed to by a pointer variable

Example:

```c
int *ptr, var1, var2;
var1 = 5;
ptr = &var1;
var2 = *ptr;
```
Pointer Arithmetic (1/2)

° Since a pointer is just a memory address, we can add to it to traverse an array.

° \texttt{ptr}+1 will return a pointer to the next array element.

° \texttt{*ptr+1} vs. \texttt{*ptr++} vs. \texttt{* (ptr+1)} ?

° What if we have an array of large structs (objects)?

  • C takes care of it: In reality, \texttt{ptr+1} doesn’t add 1 to the memory address, but rather adds the size of the array element.
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
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)
Pointer Arithmetic Peer Instruction Q

How many of the following are invalid?

I. pointer + integer  
II. integer + pointer  
III. pointer + pointer  
IV. pointer – integer  
V. integer – pointer  
VI. pointer – pointer  
VII. compare pointer to pointer  
VIII. compare pointer to integer  
IX. compare pointer to 0

A: 0  
B: 1  
C: 2  
D: 3  
E: 4  
F: 5
Signed & Unsigned Integers

- Why do computers and the C language have BOTH signed and unsigned integers?
- What would be wrong with JUST having signed integers in the computer? in C?

Given that Moore's Law doubles the number of transistors every 18 months, someone could build a decimal number computer.

- What would be the advantages and disadvantages of such a computer?
- What might a normal computer-user notice about a decimal computer?
- What might a programmer notice?
Complete the program segment below so that the contents of `intValue` and `n` before return agree with the diagram.

```
int main ( ) {
    intPtr = ___;
    n = ___;
    return 0;
}
```
Write a void function named `MultiplyBy5` that multiplies the value of an `int` variable used in the calling program by 5. Here's a framework for a main program in which `MultiplyBy5` is used:

```c
#include <stdio.h>

int main() {
    int n = 27;
    MultiplyBy5();
    printf("n = %d\n", n); /* 5n */
    return 0;
}

void MultiplyBy5(int ___) {
    ___ = ___ * 5;
}
```
How many of the following are invalid?

I. pointer + integer  
II. integer + pointer  
III. pointer + pointer  
IV. pointer – integer  
V. integer – pointer  
VI. pointer – pointer  
VII. compare pointer to pointer  
VIII. compare pointer to integer  
IX. compare pointer to 0

A: 0  
B: 1  
C: 2  
D: 3  
E: 4  
F: 5
C Pointer Dangers

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

```c
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?
More 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?

```c
void f()
{
    int* x;
    *x = 5;
}
```
Pointers & Allocation (1/2)

° After declaring a pointer:

```c
int *ptr;
```

`ptr` doesn’t actually point to anything yet. 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)
Pointing to something that already exists:

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

`var1` and `var2` have room implicitly allocated for them.
Arrays (1/5)

° Declaration:

    int ar[2];

declares a 2-element integer array.

    int ar[] = {795, 635};

declares and fills a 2-elt integer array.

° Accessing elements:

    ar[num];

returns the num^{th} element.
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:

- \texttt{ar} is a pointer
- \texttt{ar\[0\]} is the same as \texttt{*ar}
- \texttt{ar\[2\]} is the same as \texttt{*(ar+2)}
- We can use pointer arithmetic to access arrays more conveniently.

° Declared arrays are only allocated while the scope is valid

\begin{verbatim}
char *foo() {
    char string[32]; . . .;
    return string;}
\end{verbatim}

is incorrect
Arrays (4/5)

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

```c
int a[10], *p, *q, sum = 0;
...
p = &a[0]; q = &a[10];
while (p != q)
    sum = sum + *p++;
• Is this legal?
```

° C defines that one element past end of array must be a valid address, i.e., not cause an bus error or address error
Arrays (discussed in lecture, 4.5/5)

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

  - Wrong
    ```c
    int i, a[10];
    for(i = 0; i < 10; i++) {... }
    ```

  - Right
    ```c
    #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, so be careful.
Administrivia

- “I have the equivalent of 61B from my {JC, other school, etc.}”
  - You need to fill in appeal form by THIS Friday (2002-09-06)
  - 3rd floor east alcove of Soda Hall
- Read K&R 6.1–6.7, 7.8.5 for next time
- Have questions come up over the weekend?
C Strings

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

```c
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (null terminator)

```c
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}
```
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?
Next time we’ll see

° C Structures

° (very) Basic memory management
“And in Conclusion…”

- 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!”
C syntax for declaring an array is slightly different from Java:

```c
int integers[5];
char characters[10];
int moreints[] = {1, 2, 3, 4};
```

Syntax to access arrays is identical to Java:

```c
int x = integers[2];
```

Unlike Java, array bounds are NOT checked!
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.

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

int strlen(char *s) {  
    int n = 0;  
    while (s[n] != 0)  
        n++;  
    return n;  
}
```
We can use pointer arithmetic to “walk” through memory:

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

C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a char, 4 bytes for an int, etc.)
C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.

So the following are equivalent:

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