
ELEC2041

Microprocessors and Interfacing

Lecture 3: C-Language Review - II

<http://webct.edtec.unsw.edu.au/>

March, 2006

Saeid Nooshabadi

saeid@unsw.edu.au

ELEC2041 lec03-C-language-II .1

Saeid Nooshabadi

Overview

- Common Pointer Mistakes
- Pointer Arithmetic
- Arrays
- Dynamic Memory Allocation

ELEC2041 lec03-C-language-II .2

Saeid Nooshabadi

Review: Pointers in C (#1/2)

- An address refers to a particular memory location. In other words, it *points* to a memory location.
- **Pointer**: High Level Language (in this case C) way of representing a memory address.
- More specifically, a C variable can contain a pointer to something else. It actually stores the memory address that something else is stored at.

ELEC2041 lec03-C-language-II .3

Saeid Nooshabadi

Review: Pointers in C (#2/2)

- 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.

ELEC2041 lec03-C-language-II .4

Saeid Nooshabadi

Review: Pointers & Allocation (#1/2)

- After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet. We can either:

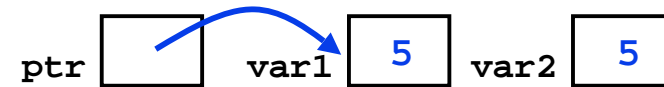
- make it point to something that already exists, or
- allocate room in memory for something new that it will point to... (next time)

Review: 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.



Review: ELEC2041 Software

- Edit Utility Tools

- Enable creation of C or assembly source programs for ARM Processor on a Linux Platform

- GNU ARM Cross Compiler and Assembler Tools:

- Enable Translation by Compilation, Assembly, and Linking of source programs into ARM object programs; Executable and Linking Format (ELF)

GNU ARM Source Level Debugger

- Enables simulation of ARM ELF programs while referencing back to the source code.

Komodo Integrated Debugger

- Enables downloading of ARM ELF code into the target ARM Processor on DSLMU Development Board
- Enables Execution and debugging of the downloaded program on the target processor on DSLMU

All Tools included in the Companion CD-ROM

C Pointer Dangers (#1/2)

- 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;
}
```

C Pointer Dangers (#2/2)

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

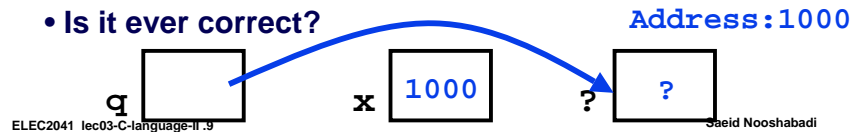
```
int x = 1000;

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?



ELEC2041 lec03-C-language-II.9

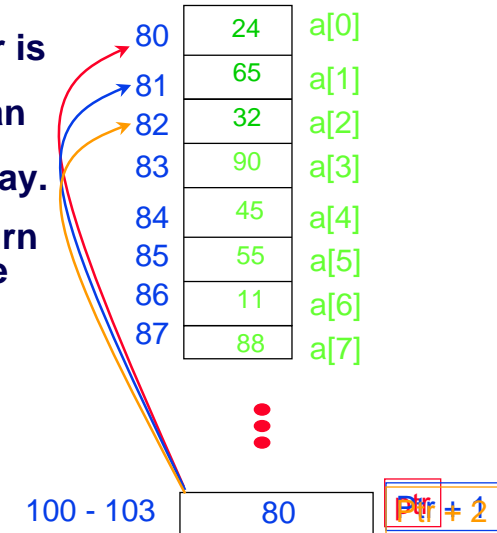
Saeid Nooshabadi

Pointer Arithmetic (#1/5)

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

- `ptr+1` will return a pointer to the next array element.

```
char a[8];
```



ELEC2041 lec03-C-language-II.10

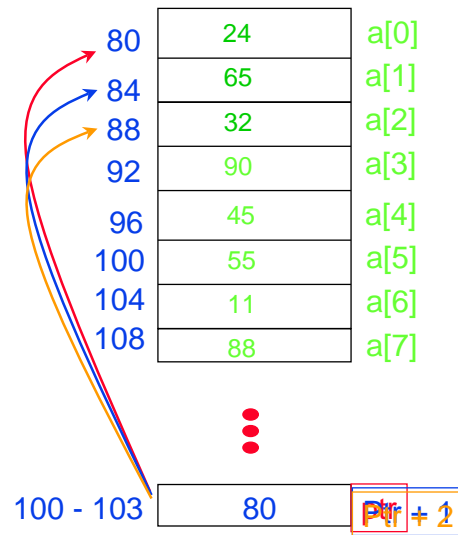
Saeid Nooshabadi

Pointer Arithmetic (#2/5)

- What about array of integers (4 Byte size).

- `ptr+1` will return a pointer to the next integer array element as well.

```
int a[8];
```



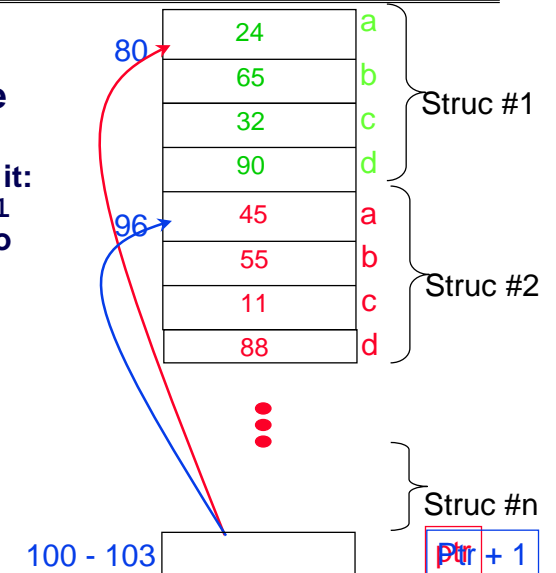
ELEC2041 lec03-C-language-II.11

Saeid Nooshabadi

Pointer Arithmetic (#3/5)

- What if we have an array of large structs?

- C takes care of it: In reality, `ptr+1` doesn't add 1 to the memory address, but rather adds the size of an array element.



ELEC2041 lec03-C-language-II.12

Saeid Nooshabadi

Pointer Arithmetic (#4/5)

- C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.
- So the following are equivalent:

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

ELEC2041 lec03-C-language-II .13

Saeid Nooshabadi

Pointer Arithmetic (#5/5)

- So what's valid pointer arithmetic?
 - Add an integer to a pointer.
 - Subtract 2 pointers (in the same array).
 - Compare pointers (<, >, etc.).
 - 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

ELEC2041 lec03-C-language-II .14

Saeid Nooshabadi

Pointer Arithmetic Peer Instruction

- Which one of the following are invalid?
 - pointer + integer OK
 - integer + pointer Not OK
 - pointer + pointer Not OK
 - pointer – integer OK
 - integer – pointer Not OK
 - pointer – pointer OK
 - compare pointer to pointer OK
 - compare pointer to integer Not OK
 - compare pointer to 0 OK

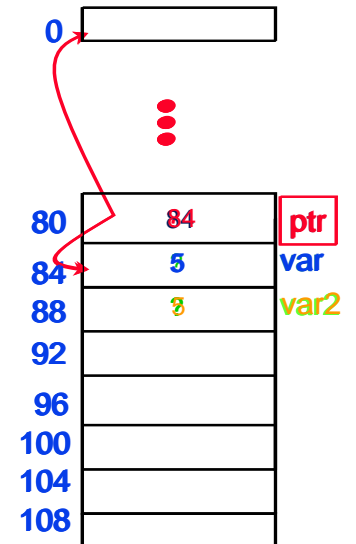
ELEC2041 lec03-C-language-II .15

Saeid Nooshabadi

Pointer Usage (#1/2)

- 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:


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



ELEC2041 lec03-C-language-II .16

Saeid Nooshabadi

Pointer Usage (#2/2)

◦ 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 element

◦ `(*p)+1` vs `*p++` vs `*(p+1)` vs `(*p)++` ?

• `x = (*p)+1` → `x = *p + 1; p = p;`



• `x = *p++` → `x = *p ; p = p + 1;`



• `x = *(p+1)` → `x = *(p+1) ; p = p;`



• `x = (*p)++` → `x = *p ; *p = *p + 1;`



Dynamic Memory Allocation (#1/4)

◦ After declaring a pointer:

```
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...**

Dynamic Memory Allocation (#2/4)

◦ Pointing to something that already exists:

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

◦ `var` and `var2` have room implicitly allocated for them.

Dynamic Memory Allocation (#3/4)

◦ To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):

```
ptr = (int *) malloc (sizeof(int));
```

- Now, `ptr` points to a space somewhere in memory of size `(sizeof(int))` in bytes.
- `(int *)` simply tells the compiler what will go into that space (called a **typecast**).

◦ `malloc` is almost never used for 1 var

```
ptr = (int *) malloc (n*sizeof(int));
```

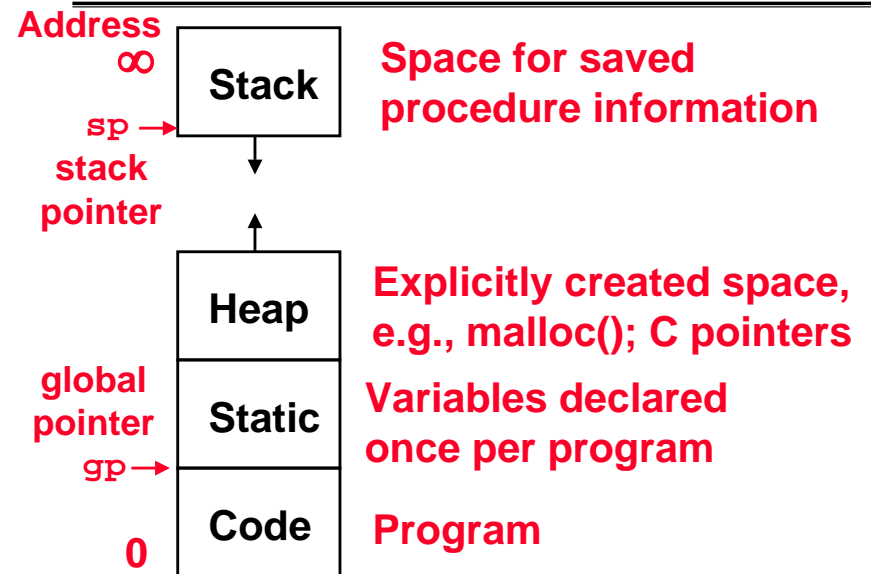
- This allocates an array of `n` integers.

Dynamic Memory Allocation (#4/4)

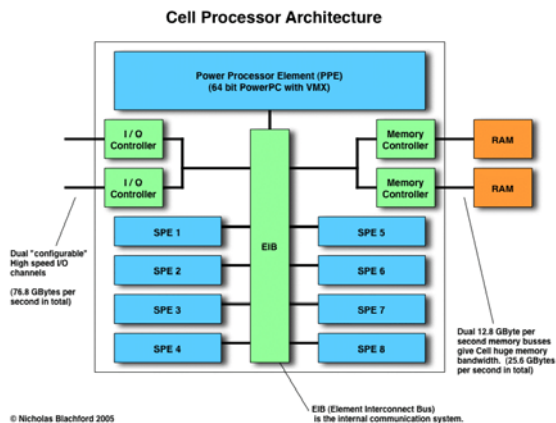
- Once `malloc()` is called, the memory location **contains garbage**, so don't use it until you've set its value.
- After dynamically allocating space, we must dynamically free it:

```
free(ptr);
```
- Use this command to clean up.

C memory allocation



An Example of embedded system: Playstation III



- The Graphic Accelerator (RSX) chip connects directly to the Cell.
- Represents 1500 person years of investment
- Delivers two teraflops of power when it's coupled to the Cell.
- Contain 300 million transistors (Pentium has 100 million)

Key to Success in ELEC2041

- Doing the Lab Experiments and Weekly Tutorial sets, and On-line quizzes on the WebCT diligently, properly and honestly.
- Last year all students who failed the course got less than 50% in quiz and less than 60% in the lab.
- GO to the Tutorial Classes:
 - We go through the problems similar to quiz from the previous week in the tutorial session.

Projects@EELC2041

◦ Dots-and-Boxes Games

- Dots and Boxes is a familiar pencil-and-paper game that most of us used to play as children. It starts from a square array of dots where each of the two players, in turn, has to connect any adjacent dots with a line, either horizontally or vertically. Whenever a player closes the fourth side of a box he initials that box and must then play again. The game goes on until all of the boxes had been initialed and the player who initialed more boxes is declared as the winner.

- Arnon Politi, in June 2004



Arrays (#1/6)

◦ Declaration:

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

declares a 2-element integer array.

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

declares and fills a 2-element integer array.

◦ Accessing elements:

```
ar[num];
```

returns the num^{th} element.

ELEEC2041 lec03-C-language-II .26

Saeid Nooshabadi

Arrays (#2/6)

◦ 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.

ELEEC2041 lec03-C-language-II .27

Saeid Nooshabadi

Arrays (#3/6)

◦ Consequences:

- `ar` is a pointer
- `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

```
int ar[8];
```

80	24	ar[0]
84	65	ar[1]
88	32	ar[2]
92	90	ar[3]
96	45	ar[4]
100	55	ar[5]
104	11	ar[6]
108	88	ar[7]

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

100 - 103 80 ar

ELEEC2041 lec03-C-language-II .28

Saeid Nooshabadi

Arrays (#4/6)

- 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 an bus error or address error

Arrays (#5/6)

- 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 (#6/6)

- 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!

Segmentation Fault vs Bus Error?

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

- **Segmentation Fault**

- 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.

- **Bus Error**

- 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.

C Strings (#1/2)

- A C String is just an array of characters:

```
char *str = "hello";
```

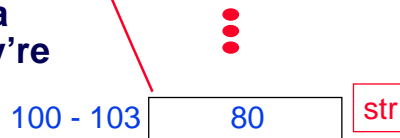
- `str` points to the 'h'

- The next five elements are:

'e', 'l', 'l', 'o', and NULL ('\0')

80	'h'	str[0]
81	'e'	str[1]
82	'l'	str[2]
83	'l'	str[3]
84	'o'	str[4]
85	0	str[5]

- **Key Detail:** A C String is always terminated with a NULL, which is why they're called null-terminated strings.



C Strings (#2/2)

- 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;
}
```

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!

Common Pointer Mistakes (#1/2)

- Declare and write:

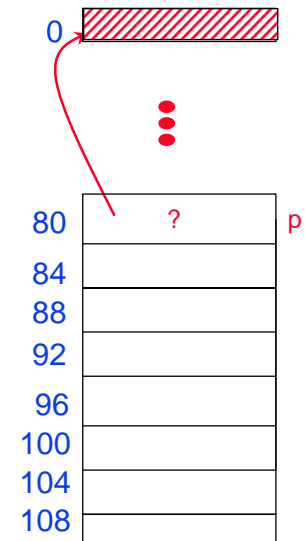
```
int *p;
```

```
*p = 10; /* WRONG */
```

- What address is in `p`?

- Answer: NULL; C defines that memory address 0 (same as NULL) is not valid to write to.

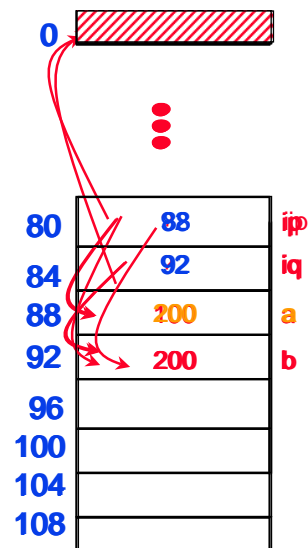
- Remember to `malloc` first.



Common Pointer Mistakes (#2/2)

° Copy pointers vs. values:

```
int *ip, *iq, a = 100, b = 200;
ip = &a; iq = &b;
*ip = *iq; /* what changed? */
ip = iq; /* what changed? */
```



Bonus Slide (near end): Arrays/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)
        n++;
    return n;
}
{
    int n = 0;
    while (s[n] != 0)
        n++;
    return n;
}
```

Could be written:
while (s[n])

Bonus Slide (near end): Pointer Arithmetic

° 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++;
    }
}
```

- ° C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a char, 4 bytes for an int, etc.)

References:

- ° Nick Parlante: [Stanford CS Education Library](http://cslibrary.stanford.edu/) (<http://cslibrary.stanford.edu/>); A Collection of very useful material including:
- ° [Essential C: \(http://cslibrary.stanford.edu/101/\)](http://cslibrary.stanford.edu/101/) A relatively quick, 45 page discussion of most of the practical aspects of programming in C.
- ° [Binky Pointer Video: \(http://cslibrary.stanford.edu/104\)](http://cslibrary.stanford.edu/104/) Silly but memorable 3 minute animated video demonstrating the basic structure, techniques, and pitfalls of using pointers.
- ° [Pointer Basics: \(http://cslibrary.stanford.edu/106\)](http://cslibrary.stanford.edu/106/) The companion text for the Binky video.
- ° [Pointers and Memory: \(http://cslibrary.stanford.edu/102\)](http://cslibrary.stanford.edu/102/) A 31 page explanation of everything you ever wanted to know about pointers and memory.
- ° [Linked List Basics: \(http://cslibrary.stanford.edu/103\)](http://cslibrary.stanford.edu/103/) A 26 page introduction to the techniques and code for building linked lists in C.

Things to Remember (#1/2)

◦ Common Pointer problems

- Declare and write
- Copy pointers vs. values

Things to Remember (#2/2)

- Use `malloc` and `free` to allow a pointer to point to something not already in a variable.
- An array name is just a pointer to the first element.
- A string is just an array of chars.