Pointers in C

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin
Modified by Diana Palsetia

Address vs. Value

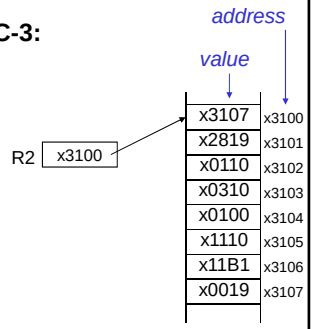
Sometimes we want to deal with the address of a memory location, rather than the value it contains

Adding a column of numbers in LC-3:

- **R2 contains address of first location**
- **Read value, add to sum, and increment R2 until all numbers have been processed**

R2 is a pointer

- **It contains the address of data**

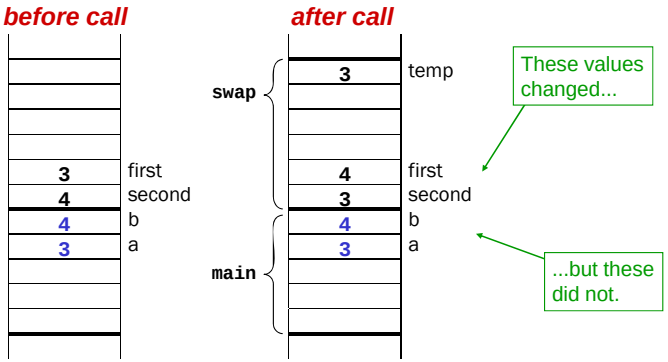


Another Need for Addresses

Consider the following function that's supposed to swap the values of its arguments.

```
void swap(int first, int second){
  int temp = first;
  first = second;
  second = temp;
}
 int main(){
   int a = 3;
   int b = 4;
   swap(a,b);
 }
```


Executing the Swap Function



Swap needs addresses of variables outside its own activation record

Pointers in C

C lets us manipulate addresses as variables and in use them in expressions.

Declaration

```
int *p;   /* p is a pointer to an int */
```

A pointer in C is always a pointer to a particular data type: `int*`, `double*`, `char*`, etc.

Operators

`*p` -- returns the value pointed to by p
a.k.a de-referencing a pointer

`&z` -- returns the address of variable z

Example

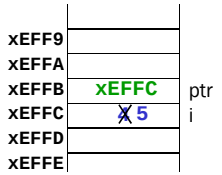
```
int main(){
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
printf("%d\n", i);

}
```

store the value 4 into the memory location associated with i

store the address of "i" into the memory location associated with ptr

read the contents of memory at the address stored in ptr and increment it by 1

Value of i will be "5", because "i" was modified indirectly via ptr



Example: LC-3 Code

```
; i is (offset 1), ptr (offset 0 i.e. the last local variable added to stack frame)
;i = 4;
      AND  R0, R0, #0   ; clear R0
      ADD  R0, R0, #4   ; put 4 in R0
      ADD  R6, R6, #-1  ;adjust stack pointer
      STR  R0, R6, #0   ; store in i on top of stack
; int *ptr = &i;
      ADD  R0, R6, #0   ; R0 = R6 + 0 (addr of i)
      ADD  R6, R6, -1   ;create space for ptr
      STR  R0, R6, #0   ; store addr of i in ptr
      AND  R5, R5, #0
      ADD  R5, R6, #0 ;main's frame pointer
;*ptr = *ptr + 1;
      LDR  R0, R5, #0   ; R0 = contents of ptr i.e. address of i
      LDR  R1, R0, #0   ; load contents (*ptr)
      ADD  R1, R1, #1   ; add one
      STR  R1, R0, #0   ; store to address of i
```


Pointers as Arguments

Passing a pointer into a function allows the function to read/change memory outside its activation record

```
void swap(int *first, int *second)
{
  int temp = *first;
  *first = *second;
  *second = temp;
}
```

Arguments are integer pointers. Caller passes addresses of variables that it wants function to change

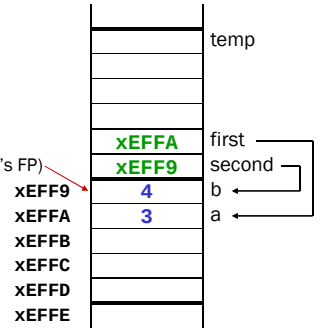
Passing Pointers to a Function

main() wants to swap the values of "a" and "b"
passes the addresses to swap():

```
swap(&a, &b);
```

Code for passing arguments:

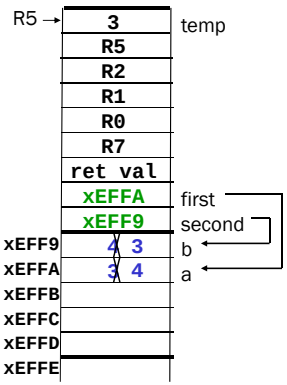
```
ADD R0, R5, #0 ; addr of b
ADD R6, R6, #-1;
STR R0, R6, #0

ADD R0, R5, #1  ; addr of a
ADD R6, R6, #-1;
STR R0, R6, #0
```



Code Using Pointers

Inside the swap() routine

```
; int temp = *first;
LDR  R0, R5, #7 ; R0=xEFFA
LDR  R1, R0, #0 ; R1=M[xEFFA]=3
STR  R1, R5, #0 ; temp=3

; *first = *second;
LDR  R1, R5, #8 ; R1=xEFF9
LDR  R2, R1, #0 ; R2=M[xEFF9]=4
STR  R2, R0, #0 ; M[xEFFA]=4

 ; *second = temp;
LDR  R2, R5, #0 ; R2=temp=3
STR  R2, R1, #0 ; M[xEFF9]=3
```



Using Arguments for Results

Pass address of variable where you want result stored

- **Example:**

```
scanf("%d %d", &data1, &data2);
```

read decimal integers into `data1` and `data2`

Bad scanf Arguments

1. Argument is not an address

```
int n = 0;
scanf("%d", n);
```

- **Will use the value of the argument as an address**
- **If you're lucky, program will crash because of trying to modify a restricted memory location (e.g., location 0)**
  - **Runtime error: Segmentation Fault**

2. Missing data argument

```
scanf("%d");
```

- **Your program will just modify an arbitrary memory location, which can cause very unpredictable behavior**
  - **Because it will get address from stack, where it expects to find first data argument**

Null Pointer

Sometimes we want a pointer that points to nothing.

In other words, we declare a pointer, but we're not ready to actually point to something yet.

```
int *p;
p = NULL;  /* p is a null pointer */
```

`NULL` is a predefined macro that contains a value that a non-null pointer should never hold.

- Often, NULL = 0, because Address 0 is not a legal address for most programs on most platforms
- Dereferencing a NULL pointer: program crash!
  - `int *p = NULL;  printf("%d", *p);` // CRASH!
  - Output: Segmentation fault

Examples: Pointer Problems

What does this do?

```
int *x;
*x = 10;
```

Answer: writes "10" into a random location in memory

What's wrong with:

```
int* func(){
  int x = 10;
  return &x;
}
```

Answer: storage for "x" disappears on return, so the returned pointer is dangling

A dangling pointer is a pointer to storage element(int, char, double etc) that is no longer allocated

Declaring Pointers

The * operator binds to the variable name, not the type

All the same:

- `int* x, y;`
- `int *x, y;`
- `int *x; int y;`

Suggested solution: Declare only one variable per line

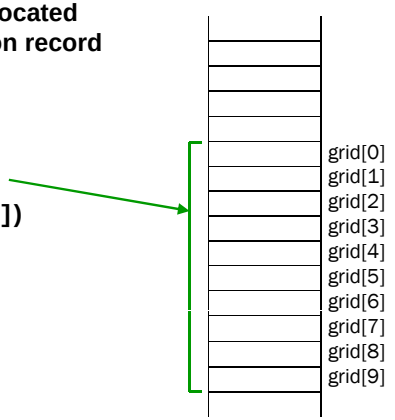
- Avoids this problem
- Easier to comment
- Clearer
- Don't worry about "saving space"

Array as a Local Variable

Array elements are allocated as part of the activation record

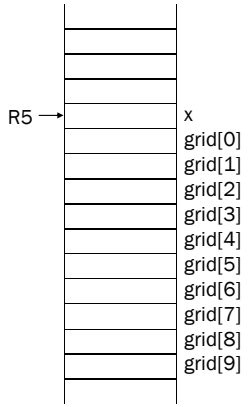
```
int grid[10];
```

First element (`grid[0]`) is at lowest address of allocated space



LC-3 Code for Array References

```
; x = grid[3] + 1
 ADD R0, R5, #1  ; R0 = &grid[0]
 LDR R1, R0, #3  ; R1 = grid[3]
 ADD R1, R1, #1  ; R1 = R1 + 1
 STR R1, R5, #0  ; x = R1

; grid[6] = 5;
 AND R0, R0, #0
 ADD R0, R0, #5  ; R0 = 5
 ADD R1, R5, #1  ; R1 = &grid[0]
 STR R0, R1, #6  ; grid[6] = R0
```



Relationship between Arrays and Pointers

An array name is essentially a pointer to the first element in the array

```
char data[10];
```

```
i.e. data = addr where first element is located
          = &data[0]
```

Example:

```
char data[10];
char *cptr;
cptr = data; /* points to data[0] */
```


Correspondence between Ptr and Array Notation

Given the declarations on the previous page, each line below gives three equivalent expressions:

cptr	data	&data[0]
(cptr + n)	(data + n)	&data[n]
*cptr	*data	data[0]
*(cptr + n)	*(data + n)	data[n]

Beware

Arrays are not the same as pointers although they may look like

what is the difference between arrays and pointers?

- **Arrays automatically allocate space, but can't be relocated or resized.**
- **Pointers must be explicitly assigned to point to allocated space but can be reassigned (i.e. pointed at different objects) at will, and have many other uses besides serving as the base of blocks of memory.**

Pointer Arithmetic: Subtraction and Equality

Nasty, but C allows it:

```
void function(int* start, int* end)
{
  int i;
  while (end - start >= 0) {
    *start = 0;
    start++;
  }
}
In function main():
int array[10];
function(&array[0], &array[9]);
```

Don't do this!

Alternative: `while (end != start) {`

- Significantly better, but still bad
- What if start is > end, or not part of same array?

More on Pointer Arithmetic

Address calculations depend on size of elements

- In our LC-3 code, we've been assuming one word per element
  - e.g., to find 4th element, we add 4 to base address
- It's ok, because we've only shown code for int, which takes up one word (equal to machine width).
- If `double`, we'd have to add 8 to find address of 4th element.

C does size calculations under the covers, depending on size of item being pointed to:

```
double x[10];
double *y = x;
*(y + 3) = 100;
```

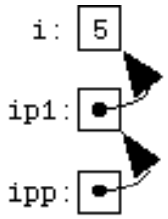
allocates 20 words (2 per element)

same as x[3] -- base address plus 6

Pointer-to-Pointer

The declaration of a pointer-to-pointer looks like
int **ipp;

Example:
int i = 5
int *ip1 = &i,
int ** ipp = &ip1;



C vs. Java

C Pointers makes it unsafe as there is no compile time checking on:

- Dereferencing a null pointer
- Having dangling pointer
- Can easily right to memory space beyond what you declared
  - No limit checking for array length

Java removes unsafe features by not

- supporting the unary '&' address operator
- supporting address arithmetic on references
  - i.e. does not allow integer values to be added/subtracted to references