Indirect Addressing

3/14/01  Lecture #15  16.070

- **Direct Addressing**: access memory location by using variable name
- **Indirect Addressing**: access memory location by using a variable whose contents contain address of desired location
- Indirect Addressing allows functions to access and modify memory locations outside of the function
- Indirect Addressing provides a method for returning multiple values from a function
Indirect Addressing Using Pointers

• Variables possess three characteristics
  ➢ Variable name
  ➢ Value (contents)
  ➢ Memory Address - identifies unique storage location in memory

• We have been accessing the contents by using the variable name

• The contents can also be accessed by referencing the memory address

• A Pointer Variable is a variable whose value is a memory address
Declaring Pointer Variables in C

• Pointer variables must be declared

• In declaration statement, pointer variable name is preceded by indirection operator, symbol *

• The type declaration must be the type of memory to which the pointer is pointing
  ➢ If the pointer is to point to an integer, the pointer is declared as type int
  ➢ If the pointer is to point to a character, the pointer is declared as type char
  ➢ Etc.

```c
double rate;
double *pointer_to_rate;

int torque_cmd;
int *ptr_torque_cmd;
```
Assigning Values to Pointers

- Once a pointer is declared, it needs to be assigned a value

- Assign as its value the address of another variable using the *address operator &*

- Translate & as "address of"

```c
int torque_cmd;
int *ptr_torque_cmd;
torque_cmd = 255;
ptr_torque_cmd = &torque_cmd;
```

- The variable `torque_cmd` now contains the integer value 255, and the variable `ptr_torque_cmd` now contains the memory address of `torque_cmd`

- The operator * extracts the value from an address

  `torque_cmd` is synonymous with `*ptr_torque_cmd`
Graphic Representation of Pointers

- Pointers are used to "point to" other memory locations
- `Ptr_torque_cmd` contains address of `torque_cmd` variable; i.e., it "points to" `torque_cmd`

![Diagram of pointer usage]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address (in hex)</th>
<th>Contents (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>torque_cmd</code></td>
<td>100</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td><code>ptr_torque_cmd</code></td>
<td>2AF</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Examples

• What are the values of the following variables?

  torque_cmd    __________

  ptr_torque_cmd _________

  &torque_cmd _________

  *ptr_torque_cmd _________

  &ptr_torque_cmd _________

• Remember:  &  means "the address of the variable"

• Remember:  *  means "the value stored at the pointer address"
More Examples

• Evaluate the following code

```c
int i;
int *iptr;

i = 4;
iptr = &i;

*iptr = *iptr + 1;
```
pointer Arithmetic - Addition

- Pointers can be operated on with integer arithmetic
  - For pointer addition, result is a new value of the pointer, which points to a new address
    \[ \text{ptr} = \text{ptr} + n; \]
  - Will point to the address of the memory location n after the original
  - Physical memory address will depend on the type of variable the pointer is associated with (C compiler knows this from the declaration)
  - C will automatically scale the pointer by: n x (# of bytes per variable)
  
<table>
<thead>
<tr>
<th>Address</th>
<th>Char (1 byte)</th>
<th>Integer (2 bytes)</th>
<th>Float/Long (4 bytes)</th>
<th>Double (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr = 1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>ptr + 1</td>
<td>1001</td>
<td>1002</td>
<td>1004</td>
<td>1008</td>
</tr>
<tr>
<td>ptr + 12</td>
<td>100C</td>
<td>1018</td>
<td>1030</td>
<td>105F</td>
</tr>
</tbody>
</table>

- The arithmetic on the pointer does not change the value stored at either the old or the new location (but does change value of pointer)
- Programmer must ensure pointer arithmetic is within a single data type
## Pointer Arithmetic - Subtraction

- Pointers can be summed and differenced
  - Result is an integer: \[ \text{distance} = \text{ptr2} - \text{ptr1}; \]
  - To make sense, pointers must point to the same type of object (char, int, etc.)
  - Result is an integer value that represents the number of objects between the two pointers
  - Example:

```c
int grade1 = 79;
int grade2 = 95;
int distance = 0;
int *ptr_grade1 = &grade1;
int *ptr_grade2 = &grade2;
...  
distance = ptr_grade2 - ptr_grade1;  
...  
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Address (in hex)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>grade1</td>
<td>100</td>
<td>79</td>
</tr>
<tr>
<td>grade2</td>
<td>110</td>
<td>95</td>
</tr>
<tr>
<td>ptr_grade1</td>
<td>33F</td>
<td>100</td>
</tr>
<tr>
<td>ptr_grade2</td>
<td>FA0</td>
<td>110</td>
</tr>
</tbody>
</table>
Methods for Passing Parameters to Functions

- Function parameters are used to pass data between functions
  - Parameter passing by value
  - Parameter passing by reference using **pointers**
    - Passing pointer parameters to a function will enable the function to modify multiple variables, instead of returning just one variable
    - Passing pointer parameters can cause the contents of the referenced variable to be modified

```
Exchanging info to/from functions
   |
   v
Accessing Global Variables
   |
   v
Pass value
   v
Pass pointer
```
Parameter Passing by Value

• Values can be passed to a function using parameters (a.k.a. arguments)

• In C, there can be any number of parameters associated with the call
  ➢ Listed as formal parameters in the definition of the called function
  ➢ Listed as actual parameters in the calling function

    float average (int num1, int num2)
    {
        float answer;
        answer = (num1 + num2) / 2;
        return answer;
    }

    int main (void)
    {
        ...
        avg = average(grade1, grade2);
        ...
    }

• Contents of variables used as actual parameters are not changed
Returning Multiple Values From Functions

• Only one value can be returned from a function, using the `return` statement

• What if the function needs to return multiple values?
  ➢ For example, write a function that accepts 3 numbers, and returns those numbers sorted from highest to lowest.
  ➢ Passing by value does not provide this capability
Passing Parameters By Reference Using Pointers

- Instead of passing a value to a function, you can pass a pointer, which references the address of a variable
- The following sample code swaps the values of two variables

```c
void swap (int *num1_ptr, int *num2_ptr)
{
    int temp;
    temp = *num1_ptr; /* temporary storage */
    *num1_ptr = *num2_ptr;
    *num2_ptr = temp;
}
```

To call this function, pass pointers as arguments:

```c
swap (ptr_grade1, ptr_grade2);
```

- This code swaps the values of the variables that num1_ptr and num2_ptr point to. The function has access to the actual variables
- Note: Global information was altered without any explicit return
Pointers and Arrays

• Pointers provide a symbolic way to use addresses

• Array name is synonymous with the address of first element of the array

Suppose we declare an array

```c
int axes[3];
```

Then axes contains the pointer to the first element of the array:

```c
axes == &axes[0]
```

These both represent the memory address of first element of `axes` array

• Array name is synonymous with the address of first element of the array

• To reference an element of an array, add to base address and dereference:

```c
axes[i] == *(axes + i)
```
Passing Arrays as Parameters

• To pass an array to a function, pass by reference -- that is, pass the memory address of the first element

• Since the name of an array represents the address of the first element, pass the array name to the function: `axes == & (axes[0])`

• When passing an array to a function, you can either
  ➢ Pass address of array and number of elements, or
  ➢ Pass starting address of array, and ending address of array
Passing Arrays as Parameters - Example 1

- Example of passing name of array and number of arguments
- Calculate the sum of an array of integers

```c
long sum (int input_array[], int n) /* pass array and size of array*/
{
    int i;
    long total = 0;
    for (i = 0; i < n, i++)
        total = total + input_array[i];
    return total;
}
...
answer = sum(axes, 3) /*specify address of array by passing name*/
```
Passing Arrays as Parameters - Example 2

- Example of passing arrays by passing starting address of array, and ending address of array

Instead of using an index to indicate which element to access, function can alter value of pointer, and point to each array element in turn

```c
int sum_using_pointers (int *start, int *end)
/* pass pointers to start of array, and to end of array*/
{
    int total = 0;
    while (start <= end)
    {
        total = total + *start;
        start++;
    }
    return total;
}
...
answer = sum_using_pointers(axes, axes+2) /* specify start and stop address of array*/
```
Pointers

• The following expressions are equivalent

\[
\text{sc\_torque} + 2 \equiv \&\text{sc\_torque}[2] \quad /*\text{ same address */}
\]
\[
*(\text{sc\_torque} + 2) \equiv \text{sc\_torque}[2] \quad /*\text{ same value */}
\]

• Indirection operator (*) has high precedence

\[
*(\text{sc\_torque} + 2) \quad /*\text{ value of 3rd elmt of sc\_torque */}
\]
\[
*\text{sc\_torque} + 2 \quad /*\text{ 2 added to value of 1st elmt */}
\]
Pointers - The Good, The Bad and the Ugly

• Pointers can add efficiency to your code

• Pointers, if not used very carefully, can get you into a lot of trouble

```c
int i = 4;
int *iptr = &i;

iptr = iptr + 2;

...

*iptr = 0;
...
```

• Don't be surprised to see Protection Faults!
Review

• Pointers are powerful. You are now armed and dangerous.
• Pass by reference vs pass by value
• PS 6 posted. Give yourself ample time to work.
• Exam 1
  ➢ Point taken off for no "return" statement will be reviewed. Give exams to me
  ➢ Proposal: Incentive formula for next exam:
    – For every point you do better on Exam 2 vs Exam 1, weight of Exam 1 will be reduced by 0.25%:
      
      \[(\text{Exam 1 weight}) = 15 - (\text{Exam 2} - \text{Exam 1}) \times 0.25\]
    – Reduced weight will be equally distributed on Exams 2 and 3