Arrays

Arrays, Argument Passing, Promotion, Demotion

Review

- Introduction to C
  - C History
  - Compiling C
  - Identifiers
  - Variables
    - Declaration, Definition, Initialization
    - Variable Types
  - Logical Operators
  - Control Structures
    - i.e. loops
- Functions and Macros
- Separate Compilation

Arrays in C

- Array - a collective name given to a group of similar quantities
  - All integers, floats, chars, etc…
  - Array of chars is called a “string”

- C Array – A block of memory locations that can be accessed using the same variable name
  - Same data type

Declaration of Arrays

- Arrays must be declared before they may be used
  - type variable_name[length];
    - Type – the variable type of the element to be stored in the array
    - Variable_name – Any name of the variable to be addressed
    - Length – computer will reserve a contiguous block of memory space according to length of array in memory

program considers the block contiguous, though the architecture may place the array in multiple pages of memory

Examples

- double height[10];
  - Type: double
  - Variable name: height
  - Length: 10

- float width[20];
- int c[9];
- char name[20];
  - Would be referred to as a string

Array Implementation in C

- Array identifier alone is a variable storing the address of the first element of the array
  - Typically dereferenced with offset to access various elements of the array for reading or modification
- First element of array is stored at position 0, second at position 1, nth at (n-1)th position
  - Accessing variable – a[n] = (n+1)th element

Initialization

- Arrays should be initialized to some value
  - Uninitialized arrays are still able to be accessed - can access garbage memory contents
- Example
  - int a[] = {10,20,30,40};
  - int a[5]={1,2,3};
    - If array size > numbers initialized, all others initialized to zero
  - int a[5] = {0};
    - Shorthand for initializing all elements to 0

Accessing Array Elements

- Accessed with a dereference and offset using the [] operator
  - After dereferenced, treated like normal variables
    - Can be read and modified like a normal variable
- Valid array access examples:
  - c[0] = 3;
  - c[3] += 5;
  - y = c[x+1] ;

Char Array

- Character arrays can be initialized using “string literals”
  - String literals are specified with double quotes and are an array of constant characters and are terminated by null character
  - A null character terminates c-style strings
    - ‘\0’ – null character
- Equivalent char arrays example:
  - Char string1[] = “first”;
  - Char string1[] = {‘f’, ‘i’, ‘r’, ‘s’, ‘t’, ‘\0’};
- Can access individual characters
  - string1[3] == ‘s’

scanf()

- Function for taking input from stdin
- Format: scanf("%s", string1);
- Function
  - Reads characters from stdin until whitespace encountered
    - Can write beyond end of array

More Char Array

- char string[5] = “hello”
  - Error: 6 characters in string literal due to null character
- Char string[10] = “hello”
  - Equivalent to “hello\0\0\0\0\0”

- Note: string=“hello” will give a syntax error

Example Program

```
#include <stdio.h>
int main(){
    char string1[20], string2[]="string";
    int I;
    printf("Enter a string: ");
    scanf("%s", string1);
    printf("string1 is: %s\nstring2 is: %s\n",string1,
       string2);
    for(i=0;string1[i]!='\0';i++)
        printf("%c",string1[i]);
    printf("\n");
    return 0;
}
```

Getting input using scanf

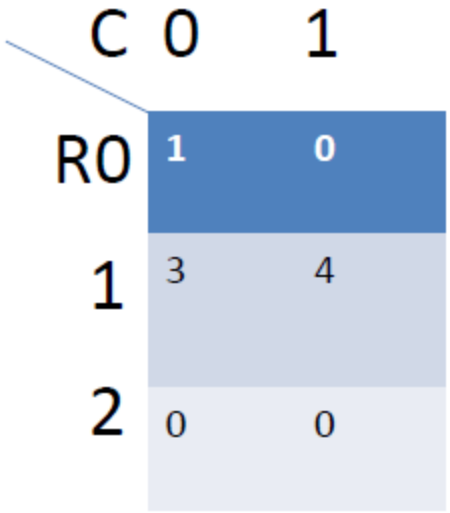
Printing strings using printf

Printing by iterating through char array

Multidimensional Arrays

- Multidimensional Array initialization:
  - Unspecified elements in given row – initialized to 0
  - Rows not given – initialized to 0

- Ex
  - int a[3][2] = {{1},{3,4}};
  - Result shown on right →



Passing Arrays to Functions

- To pass an array argument to a function specify the name of the array without any brackets
  - myFunction(myArrayName);
- Arrays are treated as “pass by reference”
  - The memory address for the array is copied and passed by value
- Name of array is the address of the first element
  - Knows where the array is stored in memory
- Modifies original memory locations

Passing Array Elements

- Array elements are passed by value
  - Original memory location is not modified
  - Ex. myFunction(myArray[3]);
    - myArray is not modified by this function

Protecting Array Elements

- const modifier will help protect contents of constant-elements by generating compiler messages
  - Example message
    - warning: passing argument 1 of ‘myFunction’ discards qualifiers from pointer target type
    - note: expected ‘char *’ but argument is of type ‘const char *’
  - Message is generated regardless of whether array is modified

Function with Const Array

- int AccessElement(const int a[], int index);
  - Coding rule: always provide const modifier in parameter types where appropriate even though it is optional
    - Prevents creating bugs
- This function would not generate a warning when called
  - Does generate an error if attempt to modify the array

Implicit Type Casting

- float f1 = 0; f2 = 1;
- int i1 = 0; i2 = 2;
- char c1 = 1; c2 = 2;

- f1 = i1/i2;
  - Int by int division, the result is cast to become a float so F0 becomes 0.0;

Explicit Type Casting

- To avoid implicit type casting compiler warnings and errors use unary cast operator
  - Unary cast operator - (type)

- Example:
  - F1 = (float)i1/(float)i2;

Demotion

- Shortening integral types
  - i.e. assigning int to char, long to int, etc…
    - Bit truncation occurs, or undefined if value cannot be stored in lower rank type
- Float to int casting attempts to truncate (remove) fractional part
  - NOT ROUNDING
  - E.g. int i = 1.5; → sets i to 1, even if i = 1.99;
- Unsigned to signed casting is particularly dangerous
  - E.g. unsigned int j = -1; //gives a very large positive number

Implicit Type Casting Functions

```
int mult(int a, int b){
    Return (a*b);
}

/* somewhere in main*/
float f0,f1,f2;
f0 = mult(f1, f2);
```

Parameter passing is like assignments, implicit casting can occur and will cause warnings

```
f0 = (float)mult( (int)f1 , (int)f2 );
```

Better to use implicit type casting