Computer Programming using C

Lecture 10: Arrays, memory allocation, and pointers to pointers

Dr Julian Miller
Room P/M/101
Ext 2383; E-mail: jfm@ohm
http://www.elec.york.ac.uk/intsys/users/jfm7

Contents

• Pointer arithmetic
• Arrays and pointers
• Arrays of pointers
• Allocating memory at run time
• Two-dimensional arrays (pointers to pointers)
Pointer arithmetic

- Pointers are *variables* that hold *addresses*
- Variables can be incremented
- Ergo: pointers must be able to be incremented. Here is how.
- Consider
  - int *p
  - p++;
- means the address held by p will have gone up *by the memory* required to hold an *integer*

Arrays and pointers

- Suppose you have declared an array
  - int a[10]
- The C compiler treats the *name* of the array *a* as the *address* of *a[0]*.
- *a[i]* is equivalent to *(a+i)
- Consider
  - int a[5],*p;
  - p=a;  //this is equivalent to p=&a[0];
  - p=a+1;  // this is equivalent to p=&a[1];
Exercise 13.2: Strings and character pointers

- Character pointers and strings are equivalent. Compare the following:

```c
void reverse_string(char string[], char revstring[]) {
    int i, j = 0, length;
    length = string_length(string);
    for (i = length - 1; i >= 0; i--) {
        revstring[j] = string[i];
        j++;
    }
    revstring[length] = '\0';
}
```

```c
void reverse_string(char *string, char *revstring) {
    int i, j = 0, length;
    length = string_length(string);
    for (i = length - 1; i >= 0; i--) {
        revstring[j] = string[i];
        j++;
    }
    revstring[length] = '\0';
}
```

Passing Arrays

- Suppose `my_array` is passed to a function
- What happens? The *address* of the initial element that is passed.
- `my_array` is treated as a pointer within the body of the function so pointer arithmetic can be done.
Pointers and Arrays: equivalents

```c
#include <stdio.h>

int main(void)
{
    int i;
    int array[5]={1,2,3,4,5};
    int *p=array;
    for (i=0;i<5;i++)
        printf("%d",array[i]);
    printf("\n");
    for (i=0;i<5;i++)
        printf("%d",*(p+i));
    printf("\n");
    for (i=0;i<5;i++)
        printf("%d",*p++);
    return 0;
}
```

Arrays of pointers

```c
int x=1,y;
int *b[10];
```

- Meaning: `b` is an array of integer pointers

```c
b[2]=&x;
```

- Meaning: third element of `b` holds the address of variable `x`

```c
y=*b[2];
```

- Meaning: integer variable `y` is assigned the contents of the variable whose address is stored in `b[2]`
Fixed dimension arrays are inconvenient

• So far we have declared arrays to be of fixed dimension, e.g.
  – `int array[100];`
• This is inconvenient as often one doesn’t know in advance how much memory to allow (e.g. the array of structures for library users)
• There is a way in C of allocating memory when the program is executed.

Memory Allocation functions

• `sizeof(my_data_type)` is an predefined operator that returns the amount of memory (in bytes) that `my_data_type` requires
  • Defined in stdlib.h
• `malloc(mysize*sizeof(mydatatype))`  
  – Reserves memory for mysize elements of size mydatatype and returns adress in memory of start of memory allocated
• `calloc(mysize, sizeof(mydatatype))`  
  – Acts exactly like malloc, except the contents of the memory allocated is set to zero
• `realloc(myarray,mysize))`  
  – This allows you to resize some memory that has previously been requested and allocated to myarray
• `free(mypointer)`  
  – Releases memory whose start address was given in mypointer
Example: allocating space for a one dimensional array

```c
int main(void) {
    int *array;
    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);
    srand(123456);
    /* make space for array[] */
    array = calloc(size, sizeof(int));
    if (array==NULL) {
        printf("ERROR. Not enough memory for array\n");
        exit(0);
    }
    fill_array(array, size);
    print_array(array, size);
    free(array);
    return 0;
}
```

The functions

```c
#include <stdio.h>
#include <stdlib.h>

void fill_array(int *array, int size);
void print_array(int *array, int size);

void fill_array(int *array, int size) {
    int i;
    /* fill array with random integers < 100 */
    for (i=0; i<size; i++)
        array[i] = rand() % 100;
}

void print_array(int *array, int size) {
    int i;
    for (i=0; i<size; i++)
        printf("%3d", array[i]);
}
Pointers to pointers

- A pointer is a variable that holds the address of another variable.
- But a pointer itself is another variable so one should be able to store the address of the pointer in a variable too. Well, we can.
- E.g.

  ```c
  int i=0,*p,**q;
  p = &i;
  *p = 3;
  q = &p; // q is given the address of pointer variable p
  **q = *p+2;
  ```

- The most common use of pointers to pointers is in the allocation of memory at execution time for multidimensional arrays. So let’s see how to do that.

---

### Allocating space for a two-dimensional array

- **array[rows-1]**
  - **array[rows-1][cols-1]**
- **array[rows-1]**
  - **array[rows-1][cols-1]**
- **array[1]**
  - **array[1][cols-1]**
- **array[0]**
  - **array[0][cols-1]**
- **array[0]**
  - **array[0][cols-1]**
- **array[1]**
  - **array[1][cols-1]**
- **array[0]**
  - **array[0][cols-1]**
- **array[0]**
  - **array[0][cols-1]**

- Point to the start of a block of memory of size `cols*sizeof(int)`
- Array of pointers
- Point to the start of a block of memory of size `rows*sizeof(int *)`
```c
int main(void)
{
    int i;
    int rows, cols;
    int **array;

    printf("Enter the number of rows of the array: ");
    scanf("%d", &rows);
    printf("Enter the number of columns of the array: ");
    scanf("%d", &cols);

    srand(123456);

    /* make space for array of pointers to int */
    array = calloc(rows, sizeof(int *));
    if (array == NULL)
    {
        printf("ERROR. Not enough memory for row pointers\n");
        exit(0);
    }

    /* make space for cols integers in each row */
    for (i = 0; i < rows; i++)
    {
        array[i] = calloc(cols, sizeof(int));
        if (array[i] == NULL)
        {
            printf("ERROR. Not enough memory for row %d\n", i);
            exit(1);
        }
    }

    fill_array(array, rows, cols);
    print_array(array, rows, cols);

    for (i = 0; i < rows; i++)
        free(array[i]);
    free(array);
    return 0;
}
```
The functions

```c
void fill_array(int **array, int rows, int cols);
void print_array(int **array, int rows, int cols);

void fill_array(int **array, int rows, int cols)
{
    int i, j;

    /* fill two dimension array with random integers < 100 */
    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            array[i][j] = rand() % 100;
}

void print_array(int **array, int rows, int cols)
{
    int i, j;

    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < cols; j++)
            printf("%3d", array[i][j]);
        printf("\n");
    }
}
```

Summary

- Pointer arithmetic
- Arrays and pointers
- Looked at how to allocate memory for arrays at run time
- Examined two dimensional arrays via pointers to pointers
- In the lab this week:
  - arrays and strings and pointers, memory allocation
- NEXT LECTURE: MONDAY WEEK 8!!!!