C Programming Language:
C Strings and Dynamic Allocation

Math 230

Assembly Language Programming
(Computer Organization)

Tuesday Jan 29, 2008

Lecture 5

Overview

- C arrays
- C strings
- Dynamic Memory Allocation

Strings and Pointer Notation(I)

- To Create Character Strings
  - Use an array of characters, or
  - Define a char pointer

```
char message1[81];
```

- //creates a pointer constant
  - the address of message1 cannot be changed
  - it always points to the first character in the array

```
char *message2;
```

- //creates a pointer to a char

Strings and Pointer Notation (II)

```
char message1[81];
```

- message1 = “this is a string”;   //INVALID!!
- char message1[81] = “this is a string”;   // valid

```
char *message2;
```

- string assignments are allowed
- message2=“this is a string”;   // valid assignment

- Strings are null terminated, ie   ‘\0’

C String Standard Functions

```
int strlen(char *string);
```

- compute the length of `string`

```
int strcmp(char *str1, char *str2);
```

- return 0 if `str1` and `str2` are identical
- return –1 or +1 based on dictionary order (like java compareTo)
- How is this different from `str1 == str2`?

```
char *strcpy(char *dst, char *src);
```

- copy the contents of string `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.

```
printf(" compare result is %d \n", strcmp("jae", "jim"));
```

Arrays of String

- An array of character pointers is quite useful for working with an array of strings
  - char *seasons[4];
    - creates an array of four elements
    - each element is a pointer to a character
    - each pointer can be assigned to point to a string

```
char *seasons[4];
seasons[0] = “Winter”;
seasons[1] = “Spring”;
seasons[2] = “Summer”;
seasons[3] = “Fall”;
```

Arrays of String

- An array of character pointers is quite useful for working with an array of strings
  - char *seasons[4];
    - creates an array of four elements
    - each element is a pointer to a character
    - each pointer can be assigned to point to a string

```
char *seasons[4];
seasons[0] = “Winter”;
seasons[1] = “Spring”;
seasons[2] = “Summer”;
seasons[3] = “Fall”;
```

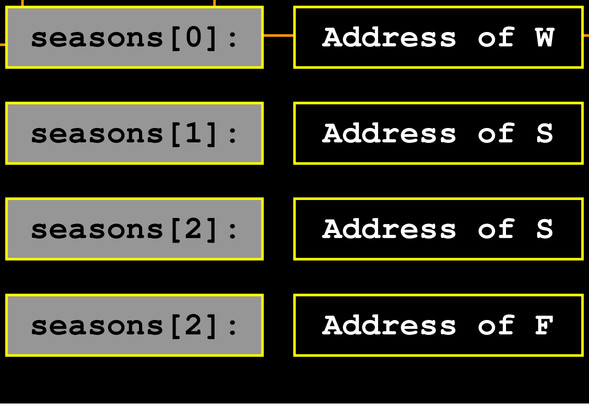
```
char *seasons[4] = {
            “Winter”,
            “Spring”,
            “Summer”,
            “Fall”};
```

Arrays of String

- An array of character pointers is quite useful for working with an array of strings
  - char *seasons[4];
    - creates an array of four elements
    - each element is a pointer to a character
    - each pointer can be assigned to point to a string

```
char *seasons[4];
seasons[0] = “Winter”;
seasons[1] = “Spring”;
seasons[2] = “Summer”;
seasons[3] = “Fall”;
```

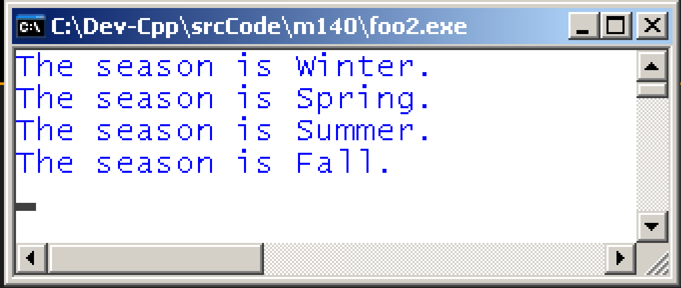
```
char *seasons[4] = {
             “Winter”,
             “Spring”,
             “Summer”,
             “Fall”};
```



seasons[0]:	Address of W
seasons[1]:	Address of S
seasons[2]:	Address of S
seasons[2]:	Address of F

Example

```
#include <stdio.h>
int main()
{
  int n;
  char *seasons[4] = {“Winter”,
                      “Spring”,
                      “Summer”,
                      “Fall”};
  for( n=0; n<4; ++n)
    printf(“\n The season is %s.”seasons[n]);
  return 0;
}
```



```
The season is Winter.
The season is Spring.
The season is Summer.
The season is Fall.
```

Command-Line Arguments

```
int main(int argc, char *argv[])
{   . . .}
```

- **`argc, argv`**
  - main accepts two arguments: `argc` and `argv`
  - `argc` is the number of arguments on the command line
  - The array `argv` contains the arguments with which the program was invoked
  - **`argv[0]`** is always the name of the executable
- See following example

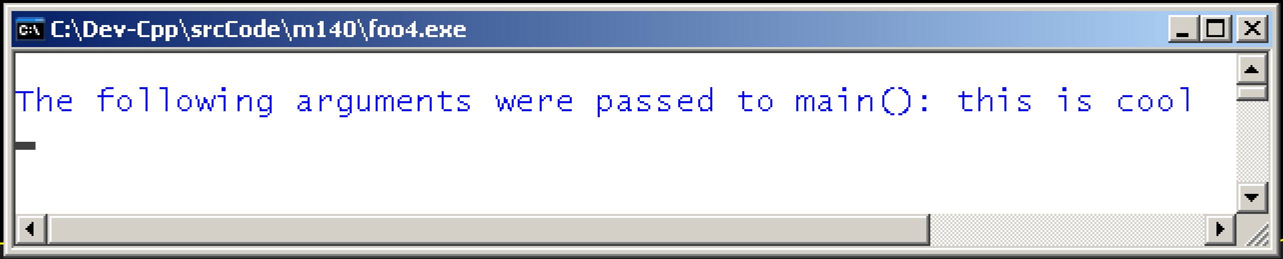
Code Example

- argc and argv

```
/* foo4.c: A program that displays command line arguments */
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;

  printf("\nThe following arguments were passed to main(): ");
  for (i = 1; i < argc; i++)
     printf("%s ", argv[i]);
  printf("\n");

  return 0;
}
```



```
The following arguments were passed to main(): this is cool
```

Command-Line Arguments

If program **foo4.exe** were run with the command line:

```
C:\foo4 I really like eating
```

then argc and argv would contain:

```
argc = 5
argv [ 0 ] = “foo4”
argv [ 1 ] = “I”
argv [ 2 ] = “really”
argv [ 3 ] = “like”
argv [ 4 ] = “eating”
```

- Also, note that since **argv** is an array of strings:

```
argv [ 3 ] [ 0 ] = ‘ l ’
argv [ 3 ] [ 1 ] = ‘ i ’
argv [ 3 ] [ 2 ] = ‘ k ’
```

Memory Management Tools

Dynamic Memory Allocation

- Dynamic memory allocation
  - the ability for a program to obtain more memory space at execution time, and to release space no longer needed
- `malloc()`, `free()`, and the operator `sizeof()`
  - essential to dynamic memory allocation

`malloc()`

- The function `malloc()` allocates storage for an object
- `malloc()` takes one argument: the size of the item to be allocated
- Memory used is taken from an area of memory called "the heap" (or free store)

malloc()

- User must give the function an indication of the amount of memory space it needs
- The user can request a specific number of bytes
- The user can request enough space for a certain type of data

- `malloc(20 * sizeof(char) )`
  - requests enough memory to store 20 characters

- `malloc( sizeof(int) )`
  - requests enough storage to store an integer

- Example:
  - `newPtr = malloc( sizeof(struct node) );`

malloc() – return value

- malloc()
  - returns the address of the first byte of storage reserved

- It returns a pointer (of type void*) to the allocated space
  - void* can be assigned to a variable of any pointer type

- If no memory is available, malloc returns a NULL pointer

Note: Older C code, some C++ compilers require you to cast the value of malloc,i.e., (int*) malloc(). Not required with C99

`free()`

- Function **`free()`** deallocates memory
  - it returns memory to the operating system

- To free the memory allocated in the previous example:
  - **`free( newPtr );`**

Code Example

Dynamic Memory Allocation

- We can make our programs more flexible if we allow the user to enter from the command line the maximum desired number of elements
- Space can be allocated at runtime using library function `malloc()`.

```
#include <stdlib.h>
main(int argc, char *argv[])
  { long int i, j, N = atoi(argv[1]);
    int *a = malloc( N*sizeof( int ) );
    if (a == NULL)
      { printf("Insufficient memory.\n"); return; }
    ...
```

Strings: Dynamic Allocation

```c
/* malloc example: string generator*/
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  int i,n;
  char * buffer;

  printf ("How long do you want the string? ");
  scanf ("%d", &i);

  buffer = malloc (i+1);
  if (buffer==NULL) exit (1);

  for (n=0; n<i; n++)
    buffer[n]=rand()%26+'a';
  buffer[i]='\0';

  printf ("Random string: %s\n",buffer);
  free (buffer);

  return 0;
}
```

Dynamic Array Allocation, Addressing

- Recall that &a[0] == a => a pointer constant

```
int a[20];
```

```
int* a;
a = malloc(20 * sizeof(int));
```

- C will do pointer math. Note:

```
(a+2) → &a[0] + 2*sizeof(a[0])
```