CMSC 313 COMPUTER ORGANIZATION & ASSEMBLY LANGUAGE PROGRAMMING

LECTURE 14, SPRING 2013

TOPICS TODAY

- Recap arrays vs pointers
- Characters & strings & pointers (Oh My!)
- Structs & pointers
- `const` pointers
- Project 5

RECAP
ARRAYS VS. POINTERS

C Parameter Passing Notes

- We'll say *formal parameter* vs *actual parameter*.
    - Formal parameters are place holders in function definition.
    - Actual parameters (aka arguments) actually have a value.
- In C, all parameters are passed by value.
- Parameter passing by reference is simulated by passing the *address* of the variable.

```
scanf("%d", &n) ;
```

- Array names represent the address of the array. In effect, arrays are passed by reference.

```
int UpdateArray (int A[], int n) {
   A[0] += 5 ;
   ...
```

Printing an Array

- The code below shows how to use a parameter array name as a pointer.

```
void printGrades( int grades[ ], int size )
{
  int i;
  for (i = 0; i < size; i++)
   printf( "%d\n", *grades );
   ++grades;
}
```

- What about this prototype?

```
void printGrades( int *grades, int size );
```

Passing Arrays

- Arrays are passed "by reference" (its address is passed by value):

```
int sumArray( int A[], int size) ;
```

is equivalent to

```
int sumArray( int *A, int size) ;
```

- Use `A` as an array name or as a pointer.
- The compiler always sees `A` as a pointer. In fact, any error messages produced will refer to `A` as an `int *`

sumArray

```
int sumArray( int A[ ], int size)
{
   int k, sum = 0;
   for (k = 0; k < size; k++)
        sum += A[ k ];
   return sum;
}
```

sumArray (2)

```
int sumArray( int A[ ], int size)
{
   int k, sum = 0;
   for (k = 0; k < size; k++)
        sum += *(A + k);
   return sum;
}

int sumArray( int A[ ], int size)
{
   int k, sum = 0;
   for (k = 0; k < size; k++)
   }
        sum += *A;
        ++A;
   }
   return sum;
}
```

CHARACTERS
& STRINGS
& POINTERS

Strings revisited

Recall that a string is represented as an array of characters terminated with a null (\0) character.

As we've seen, arrays and pointers are closely related. A string <u>constant</u> may be declared as either

```
char[ ] or  char *
```

as in

```
char hello[ ] = "Hello Bobby";
```

or (almost) equivalently

```
char *hi = "Hello Bob";
```

A `typedef` could also be used to simplify coding

```
typedef char* STRING;
STRING hi = "Hello Bob";
```

Arrays of Pointers

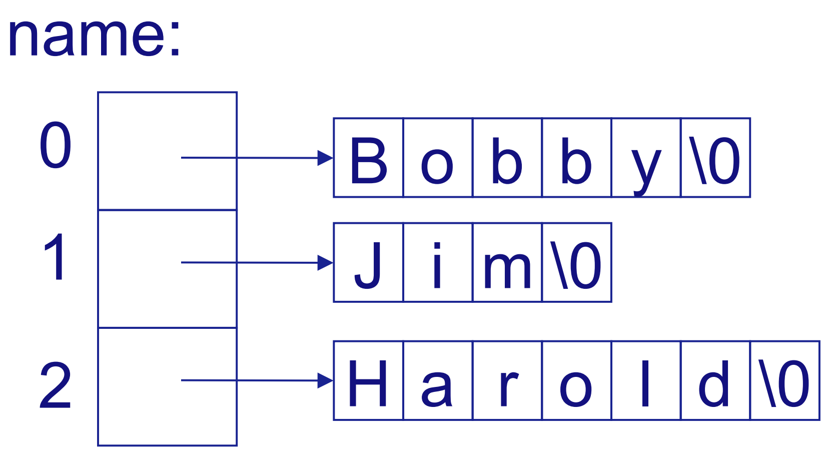
Since a pointer is a variable type, we can create an array of pointers just like we can create any array of any other type.

Although the pointers may point to any type, the most common use of an array of pointers is an array of `char*` to create an array of strings.

Boy's Names

A common use of an array of pointers is to create an array of strings. The declaration below creates an initialized array of strings `(char *)` for some boys names. The diagram below illustrates the memory configuration.

```
char *name[] = { "Bobby", "Jim", "Harold" };
```



```c
/* File: array_of_strings1.c
   A two-dimensional array of char
*/

#include <stdio.h>

int main() {
   char months[][4] =
     {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
      "Aug", "Sep", "Oct", "Nov", "Dec" } ;

   int i ;

   printf("*** Original ***\n") ;
   for (i = 0 ; i < 12 ; i++) {
      printf("%s\n", months[i]) ;
   }

   months[0][1] = 'u' ;

   printf("\n*** Modified ***\n") ;
   for (i = 0 ; i < 12 ; i++) {
      printf("%s\n", months[i]) ;
   }

   return 0 ;
}
```

```
*** Original ***
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec

*** Modified ***
Jun
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
```

```c
/* File: array_of_strings2.c
   A two-dimensional array of char.  Using a typedef.
*/

#include <stdio.h>

typedef char Acronym[4] ;

int main() {
   Acronym months[] =
     {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
      "Aug", "Sep", "Oct", "Nov", "Dec" } ;

   int i ;

   printf("*** Original ***\n") ;
   for (i = 0 ; i < 12 ; i++) {
      printf("%s\n", months[i]) ;
   }

   months[0][1] = 'u' ;

   printf("\n*** Modified ***\n") ;
   for (i = 0 ; i < 12 ; i++) {
      printf("%s\n", months[i]) ;
   }

   return 0 ;
}
```

```
*** Original ***
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec

*** Modified ***
Jun
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
```

```c
/* File: array_of_strings3.c

   A one-dimensional array of char pointers.
*/


#include <stdio.h>

int main() {

   char *name[] = { "Bobby", "Jim", "Harold" } ;

   printf("Three amigos: %s, %s and %s.\n",
      name[0], name[1], name[2]) ;

   // can point to another name this way.
   name[2] = "Jimbo" ;

   printf("Three good ole boys: %s, %s and %s.\n",
      name[0], name[1], name[2]) ;

   return 0 ;
}
```

```
Script started on Thu Oct 18 08:20:48 2012

River[7]% gcc -Wall array_of_strings3.c
River[8]% ./a.out
Three amigos: Bobby, Jim and Harold.
Three good ole boys: Bobby, Jim and Jimbo.
River[9]% exit

Script done on Thu Oct 18 08:21:02 2012
```

```c
/* File: array_of_strings4.c

   An array of char pointers.
*/

#include <stdio.h>

int main() {

   char *name[] = { "Bobby", "Jim", "Harold" } ;

   printf("Three amigos: %s, %s and %s.\n",
      name[0], name[1], name[2]) ;

   // Can point to a different name this way
   name[2] = "Jimbo" ;

   printf("Three good ole boys: %s, %s and %s.\n",
      name[0], name[1], name[2]) ;

   printf("Last 2 letters of \"Jimbo\" are '%c' and '%c'\n",
      name[2][3], name[2][4] ) ;

   // Change "Jimbo" to "Jimmy"??
   // These two assignments cause segmentation faults
   name[2][3] = 'm' ;
   name[2][4] = 'y' ;

   return 0 ;
}
```

```
Script started on Thu Oct 18 08:21:05 2012

River[10]% gcc -Wall array_of_strings4.c
River[11]% ./a.out
Three amigos: Bobby, Jim and Harold.
Three good ole boys: Bobby, Jim and Jimbo.
Last 2 letters of "Jimbo" are 'b' and 'o'
Bus error
River[12]% exit

Script done on Thu Oct 18 08:21:21 2012
```

```c
/* File: array_of_strings5.c

   A one-dimensional array of char pointers.
*/

#include <stdio.h>

int main() {

   char *name[] = { "Bobby", "Jim", "Harold" } ;

   // An honest to goodness array of char
   char jimbo[] = "Jimbo" ;

   printf("Three amigos: %s, %s and %s.\n",
      name[0], name[1], name[2]) ;

   // point to array of char named jimbo
   name[2] = jimbo ;

   printf("Three good ole boys: %s, %s and %s.\n",
      name[0], name[1], name[2]) ;

   printf("Last 2 letters of \"Jimbo\" are '%c' and '%c'\n",
      name[2][3], name[2][4] ) ;

   // Can use two-dimensional array syntax to rename Jimbo to Jimmy
   name[2][3] = 'm' ;
   name[2][4] = 'y' ;

   printf("Jimbo changed his name to %s\n", name[2]) ;
   printf("Jimbo's new name is %s\n", jimbo) ;

   return 0 ;
}
```

```
Script started on Thu Oct 18 08:21:25 2012

River[13]% gcc -Wall array_of_strings5.c
River[14]% ./a.out
Three amigos: Bobby, Jim and Harold.
Three good ole boys: Bobby, Jim and Jimbo.
Last 2 letters of "Jimbo" are 'b' and 'o'
Jimbo changed his name to Jimmy
Jimbo's new name is Jimmy
River[15]% exit

Script done on Thu Oct 18 08:21:39 2012
```

Command Line Arguments

Command line arguments:

```
./a.out breakfast lunch dinner
```

These arguments are passed to your program as parameters to main.

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

`argc` is the number of command line arguments

`argv` is an array of `argc` strings

`argv[0]`is always the name of your executable program.

The rest of `argv[]` are the remaining strings on the command line.

Command Line Arguments (2)

Example, with this command at the Linux prompt:

```
myprog hello world 42
```

we get

```
argc = 4
argv[0] = "myprog"
argv[1] = "hello"
argv[2] = "world"
argv[3] = "42"
```

Note: argv[3] is a string NOT an integer. Convert using `atoi()`:

```
int answer = atoi( argv[3] );
```

STRUCTS & POINTERS

Reminder

You can’t use a pointer until it points to something
Just declaring a variable to be a pointer is not enough

```
int *name; /* pointer declaration */
int age = 42;

*name += 12;
printf(“My age is %d\n”, *name);
```

Pointers to Pointers

A pointer may point to another pointer.

Consider the following declarations

```
int age = 42;       /* an int */
int *pAge = &age;   /* a pointer to an int */
int **ppAge = &pAge;/* pointer to pointer to int */
```

Draw a memory picture of these variable and their relationships

What type and what value do each of the following represent?

```
age, pAge, ppAge, *pAge, *ppAge, **ppAge
```

pointers2pointer.c

```
int main ( )
{
   /* a double, a pointer to double,
   ** and a pointer to a pointer to a double */
   double gpa = 3.25, *pGpa, **ppGpa;

   /* make pgpa point to the gpa */
   pGpa = &gpa;

   /* make ppGpa point to pGpa (which points to gpa) */
   ppGpa = &pGpa;

   // what is the output from this printf statement?
   printf( "%0.2f, %0.2f, %0.2f", gpa, *pGpa, **ppGpa);

   return 0;
}
```

Pointers to `struct`

```
typedef struct student {
   char name[50];
   char major [20];
   double gpa;
} STUDENT;

STUDENT bob = {"Bob Smith", "Math", 3.77};
STUDENT sally = {"Sally", "CSEE", 4.0};
STUDENT *pStudent;   /* pStudent is a "pointer to struct student" */

pStudent = &bob;     /* make pStudent point to bob */

/* use -> to access the members */
printf ("Bob's name: %s\n", pStudent->name);
printf ("Bob's gpa : %f\n", pStudent->gpa);

/* make pStudent point to sally */
pStudent = &sally;
printf ("Sally's name: %s\n", pStudent->name);
printf ("Sally's gpa: %f\n", pStudent->gpa);
```

Pointer in a struct

The data member of a `struct` can be a pointer

```
#define FNSIZE 50
#define LNSIZE 40
typedef struct name
{
   char first[ FNSIZE + 1 ];
   char last [ LNSIZE + 1 ];
} NAME;

typedef struct person
{
   NAME *pName;   // pointer to NAME struct
   int age;
   double gpa;
} PERSON;
```

Pointer in a struct (2)

Given the declarations below, how do we access bob's name, last name, and first name?

Draw a picture of memory represented by these declarations

```
NAME bobsName = {"Bob", "Smith"};
PERSON bob;
bob.age = 42;
bob.gpa = 3.4;
bob.pName = &bobsName;
```

Self-referencing structs

Powerful data structures can be created when a data member of a `struct` is a `pointer to a struct` of the same kind.

The simple example on the next slide illustrates the technique.

teammates.c

```
typedef struct player
{
   char name[20];
   struct player *teammate;/* can’t use TEAMMATE yet */
} TEAMMATE;

TEAMMATE *team, bob, harry, john;
team = &bob;                /* first player */

strncpy(bob.name, “bob”, 20);
bob.teammate = &harry;      /* next teammate */

strncpy(harry.name, “harry”, 20);
harry.teammate = &john;     /* next teammate */

strncpy(john.name, “bill”, 20);
john.teammate = NULL:       /* last teammate */
```

Adapted from Dennis Frey CMSC 313 Spring 2011

teammates.c (cont’d)

```
/* typical code to print a (linked) list */

/* follow the teammate pointers until
** NULL is encountered */

// start with first player
TEAMMATE *t = team;

// while there are more players...
while (t != NULL)
{
   printf(“%s\n”, t->name);

   // next player
   t = t->teammate;
}
```

CONST POINTERS

CONST POINTERS

4 ways to declare pointers in combination with `const`:

```
int *ptr ;                // no restriction

const int *ptr ;          // can't change *ptr

int * const ptr ;         // can't change ptr

const int * const ptr ; // can't change either
```

Mostly used with formal parameters.

```
 1   /* File: constptr1.c
 2
 3      Demonstrating const pointers.
 4   */
 5
 6
 7   void foo(int a, const int b) {
 8
 9      a = 3 ;
10      b = 4 ;    // error!
11
12      return ;
13   }
14
15
16   int main() {
17      int n, m ;
18
19      foo(n, m) ;
20
21      return 0 ;
22   }
```

```
Script started on Thu Oct 18 07:29:10 2012

River[46]% gcc -Wall constptr1.c
constptr1.c: In function ‘foo’:
constptr1.c:10: error: assignment of read-only location
River[47]% exit
exit

Script done on Thu Oct 18 07:29:18 2012
```

```
 1   /* File: constptr2.c
 2
 3      Demonstrating const pointers.
 4   */
 5
 6
 7   void foo(int *ptr1, const int *ptr2) {
 8
 9      *ptr1 = 3 ;
10      *ptr2 = 5 ;   // error ;
11
12      return ;
13   }
14
15
16   int main() {
17      int n, m ;
18
19      foo(&n, &m) ;
20
21      return 0 ;
22   }
```

```
Script started on Thu Oct 18 07:44:26 2012

River[48]% gcc -Wall constptr2.c
constptr2.c: In function ‘foo’:
constptr2.c:10: error: assignment of read-only location
River[49]% exit

Script done on Thu Oct 18 07:44:35 2012
```

```c
/* File: constptr3.c

   Demonstrating const pointers.
*/


void foo(int *ptr1, const int *ptr2) {
   int i ;

   *ptr1 = 3 ;
   // *ptr2 = 5 ;   // error ;
   ptr2 = &i ;      // is allowed

   return ;
}


int main() {
   int n, m ;

   foo(&n, &m) ;

   return 0 ;
}
```

```
Script started on Thu Oct 18 07:48:17 2012

River[54]% gcc -Wall constptr3.c
River[55]% exit

Script done on Thu Oct 18 07:48:26 2012
```

```
 1   /* File: constptr4.c
 2
 3      Demonstrating const pointers.
 4   */
 5
 6
 7   void foo(int *ptr1, const int * const ptr2) {
 8      int i ;
 9
10      *ptr1 = 3 ;
11      *ptr2 = 5 ;   // error
12      ptr2 = &i ;   // also an error
13
14      return ;
15   }
16
17
18   int main() {
19      int n, m ;
20
21      foo(&n, &m) ;
22
23      return 0 ;
24   }
```

```
Script started on Thu Oct 18 07:48:30 2012

River[56]% gcc -Wall constptr4.c
constptr4.c: In function ‘foo’:
constptr4.c:11: error: assignment of read-only location
constptr4.c:12: error: assignment of read-only location
River[57]% exit

Script done on Thu Oct 18 07:48:36 2012
```

NEXT TIME

- Dynamic memory allocation