CMSC 313
COMPUTER ORGANIZATION
&
ASSEMBLY LANGUAGE
PROGRAMMING

LECTURE 14, FALL 2012

TOPICS TODAY

- Midterm exam topics
- Recap arrays vs pointers
- Characters & strings & pointers (Oh My!)
- Structs & pointers

MIDTERM EXAM TOPICS

MIDTERM EXAM

- **Tuesday, October 23**
- **In Class**
- **No Calculators, cell phones, electronics, ...**

MIDTERM FORMAT

- Multiple Choice
- Short responses (e.g., base conversion)
- Trace assembly language program
- Write assembly language program
- Full text of `toupper.asm` available

MIDTERM TOPICS

- **Base Conversion**
- **Data Representation**
  - negative numbers: 2's complement, 1's complement, signed magnitude
  - ASCII
  - little endian vs big endian
- **Intel CPU**
  - Registers
  - Addressing modes
  - Flags
  - Common instructions

Common Instructions

- **Basic Instructions**
  - ADD, SUB, INC, DEC, MOV, NOP
- **Branching Instructions**
  - JMP, CMP, J*cc*
- **More Arithmetic Instructions**
  - NEG, MUL, IMUL, DIV, IDIV
- **Logical (bit manipulation) Instructions**
  - AND, OR, NOT, SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR
- **Subroutine Instructions**
  - PUSH, POP, CALL, RET

MIDTERM TOPICS (CONT'D)

- **Comparison & conditional jump instructions**
  - signed vs unsigned conditional jumps (e.g. ja vs jg)
- **NASM**
  - How to assemble
  - `.data`, `.bss`, `.text` sections
  - `dd`, `dw`, `db`, `resd`, `resw`, `resb` directives
  - `%define`
- **System calls for read & write**
- **Separate compilation, linking & loading**
- **Interrupts (general principles)**

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;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

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 constant 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";
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Arrays of Pointers

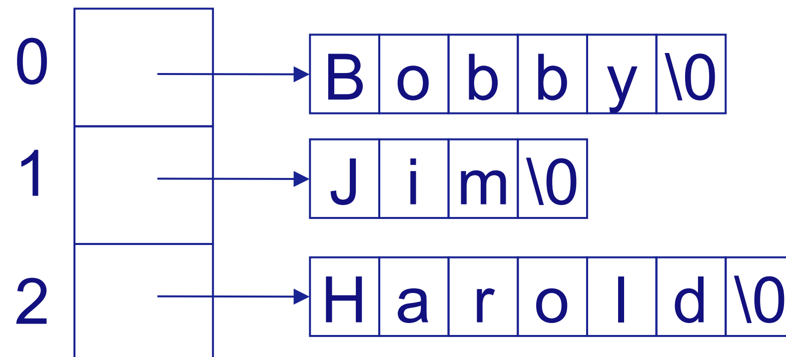
Since a pointer is a variable type, we can create an array of pointers just like we can create any array of any other type.

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" };
```



Adapted from Dennis Frey CMSC 313 Spring 2011

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);
```

Adapted from Dennis Frey CMSC 313 Spring 2011

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;
}
```

NEXT TIME

- **Dynamic memory allocation**