CMSC 313
COMPUTER ORGANIZATION
&
ASSEMBLY LANGUAGE PROGRAMMING

LECTURE 13

TOPICS TODAY

- Pointer Basics
- Pointers & Arrays
- Pointers & Strings
- Pointers & Structs

POINTER BASICS

Java Reference

- In Java, the name of an object is a reference to that object. Here ford is a reference to a Truck object. It contains the memory address at which the Truck object is stored.

```
Truck ford = new Truck( );
```

- The syntax for using the reference is pretty simple. Just use the “dot” notation.

```
ford.start( );
ford.drive( 23 );
ford.turn (LEFT);
```

Adapted from Dennis Frey CMSC 313 Spring 2011

What is a pointer ?

- pointer = memory address + type

- C pointers vs Java references
  - A pointer can contain the memory address of any variable type (Java references only refer to objects)
    - A primitive (int, char, float)
    - An array
    - A struct or union
    - Dynamically allocated memory
    - Another pointer
    - A function
  - There’s a lot of syntax required to create and use pointers

Adapted from Dennis Frey CMSC 313 Spring 2011

Why Pointers?

- They allow you to refer to large data structures in a compact way
- They facilitate sharing between different parts of programs
- They make it possible to get new memory dynamically as your program is running
- They make it easy to represent relationships among data items.

Adapted from Dennis Frey CMSC 313 Spring 2011

Pointer Caution

- Undisciplined use can be confusing and thus the source of subtle, hard-to-find bugs.
  - Program crashes
  - Memory leaks
  - Unpredictable results

- About as "dangerous" as memory addresses in assembly language programming.

Adapted from Dennis Frey CMSC 313 Spring 2011

C Pointer Variables

- General declaration of a pointer

```
type *nameOfPointer ;
```

- Example:

```
int *ptr1 ;
```

- Notes:
  - `*` = dereference
  - "if I dereference `ptr1`, I have an `int`"
  - name of pointer variable should indicate it is a pointer
  - here x is pointer, y is NOT:

```
int *x, y;
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Pointer Operators

*** = dereference**

The * operator is used to define pointer variables and to dereference a pointer. “Dereferencing” a pointer means to use the value of the pointee.

& = address of

The & operator gives the address of a variable.

Recall the use of & in scanf( )

Pointer Examples

```
int x = 1, y = 2 ;
int *ip ;   /* pointer to int */

ip = &x ;
y = *ip ;
*ip = 0 ;
*ip = *ip + 10 ;

*ip += 1 ;
(*ip)++ ;
ip++ ;
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Pointer and Variable types

The type of a pointer and its pointee must match

```
int a = 42;
int *ip;
double d = 6.34;
double *dp;

ip = &a;  /* ok -- types match */
dp = &d;  /* ok */
ip = &d;  /* compiler error -- type mismatch */
dp = &a;  /* compiler error */
```

Adapted from Dennis Frey CMSC 313 Spring 2011

More Pointer Code

```
int a = 1, *ptr1;

ptr1 = &a ;
printf("a = %d, &a = %p, ptr1 = %p, *ptr1 = %d\n",
           a, &a, ptr1, *ptr1) ;

*ptr1 = 35 ;

printf(“a = %d, &a = %p, ptr1 = %p, *ptr1 = %d\n", a,
           &a, ptr1, *ptr1) ;
```

Adapted from Dennis Frey CMSC 313 Spring 2011

NULL

- NULL is a special value which may be assigned to a pointer
- NULL indicates that a pointer points to nothing
- Often used when pointers are declared

```
   int *pInt = NULL;
```

- Used as return value to indicate failure

```
int *myPtr;
myPtr = myFunction( );
if (myPtr == NULL){
   /* something bad happened */
}
```

- **Dereferencing a pointer whose value is NULL will result in program termination.**

Adapted from Dennis Frey CMSC 313 Spring 2011

Pointers and Function Arguments

- Since C passes all primitive function arguments “by value”.

```
/* version 1 of swap */
void swap (int a, int b)
{
     int temp;
     temp = a;
     a = b;
     b = temp;
}

/* calling swap from somewhere in main() */
int x = 42, y = 17;
swap( x, y );
printf(“%d, %d\n”, x, y);   // what does this print?
```

Adapted from Dennis Frey CMSC 313 Spring 2011

A better swap( )

```
/* pointer version of swap */
void swap (int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}


/* calling swap from somewhere in main( ) */
int x = 42, y = 17;
swap( &x, &y );
printf(“%d, %d\n”, x, y);  // what does this print?
```

Adapted from Dennis Frey CMSC 313 Spring 2011

More Pointer Function Parameters

- Passing the address of variable(s) to a function can be used to have a function “return” multiple values.

- The pointer arguments point to variables in the calling code which are changed (“returned”) by the function.

ConvertTime.c

```
void convertTime (int time, int *pHours, int *pMins)
{
   *pHours = time / 60;
   *pMins = time % 60;
}

int main( )
{
   int time, hours, minutes;
   printf("Enter a time duration in minutes: ");
   scanf ("%d", &time);
   convertTime (time, &hours, &minutes);
   printf("HH:MM format: %d:%02d\n", hours, minutes);
   return 0;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

An Exercise

- What is the output from this code?

```
void myFunction (int a, int *b)
{
   a = 7 ;
   *b = a ;
   b = &a ;
   *b = 4 ;
   printf("%d, %d\n", a, *b) ;
}

int main()
{
   int m = 3, n  = 5;
   myFunction(m, &n) ;
   printf("%d, %d\n", m, n) ;
   return 0;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Pointers to `struct`

```
/* define a struct for related student data */
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};


/* pStudent is a "pointer to struct student" */
STUDENT *pStudent;

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

Adapted from Dennis Frey CMSC 313 Spring 2011

Pointers to struct (2)

```
/* pStudent is a "pointer to struct student" */
STUDENT *pStudent;

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

printf ("Bob's name: %s\n", (*pStudent).name);
printf ("Bob's gpa : %f\n", (*pStudent).gpa);

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

Pointer to struct for functions

```
void printStudent(STUDENT *studentp)
{
   printf(“Name : %s\n”, studentp->name);
   printf(“Major: %s\n”, studentp->major);
   printf(“GPA  : %4.2f”, studentp->gpa);
}
```

Passing a pointer to a struct to a function is more efficient than passing the struct itself. Why is this true?

Adapted from Dennis Frey CMSC 313 Spring 2011

POINTERS & ARRAYS

Pointers and Arrays

- In C, there is a strong relationship between pointers and arrays.
- The declaration `int a[10];` defines an array of 10 integers.
- The declaration `int *p;` defines `p` as a "`pointer to an int`".
- The assignment `p = a;` makes `p` an alias for the array and sets `p` to point to the first element of the array. (We could also write `p = &a[0];`)
- We can now reference members of the array using either `a` or `p`

```
a[4] =9;

p[3] = 7;

int x = p[6] + a[4] * 2;
```

Adapted from Dennis Frey CMSC 313 Spring 2011

More Pointers and Arrays

- The name of an array is equivalent to a pointer to the first element of the array and vice-versa.
- Therefore, if `a` is the name of an array, the expression `a[ i ]` is equivalent to `*(a + i)`.
- It follows then that `&a[ i ]` and `(a + i)` are also equivalent. Both represent the address of the `i-th` element beyond `a`.
- On the other hand, if `p` is a pointer, then it may be used with a subscript as if it were the name of an array.

  `p[ i ]` is identical to `*(p + i)`

In short, an array-and-index expression is equivalent to a pointer-and-offset expression and vice-versa.

Adapted from Dennis Frey CMSC 313 Spring 2011

So, what’s the difference?

- If the name of an array is synonymous with a pointer to the first element of the array, then what’s the difference between an array name and a pointer?

- An array name can only “point” to the first element of its array. It can never point to anything else.

- A pointer may be changed to point to any variable or array of the appropriate type

Array Name vs Pointer

```
int g, grades[ ] = {10, 20, 30, 40 }, myGrade = 100, yourGrade = 85, *pGrade;

/* grades can be (and usually is) used as array name */
for (g = 0; g < 4; g++)
      printf(“%d\n” grades[g]);

/* grades can be used as a pointer to its array if it doesn’t change*/
for (g = 0; g < 4; g++)
      printf(“%d\n” *(grades + g);

/* but grades can’t point anywhere else */
grades = &myGrade;                  /* compiler error */

/* pGrades can be an alias for grades and used like an array name */
pGrades = grades;                   /* or pGrades = &grades[0]; */
for( g = 0; g < 4; g++)
      printf( “%d\n”, pGrades[g]);

/* pGrades can be an alias for grades and be used like a pointer that changes */
for (g = 0; g < 4; g++)
      printf(“%d\n” *pGrades++);

/* BUT, pGrades can point to something else other than the grades array */
pGrades = &myGrade;
printf( “%d\n”, *pGrades);
pGrades = &yourGrade;
printf( “%d\n”, *pGrades);
```

Adapted from Dennis Frey CMSC 313 Spring 2011

More Pointers & Arrays

- If `p` points to a particular element of an array, then `p + 1` points to the next element of the array and `p + n` points n elements after `p`.
- The meaning a “adding 1 to a pointer” is that

  `p + 1` points to the next element in the array, REGARDLESS of the type of the array.

Pointer Arithmetic

- If `p` is an alias for an array of ints, then `p[ k ]` is the `k`-th int and so is `*(p + k)`.

- If `p` is an alias for an array of doubles, then `p[ k ]` is the `k`-th double and so is `*(p + k)`.

- Adding a constant, `k`, to a pointer (or array name) actually adds `k * sizeof(pointer type)` to the value of the pointer.

- This is one important reason why the type of a pointer must be specified when it's defined.

Adapted from Dennis Frey CMSC 313 Spring 2011

Pointer Gotcha

- But what if `p` isn’t the alias of an array?
- Consider this code.

```
int a = 42;
int *p = &a;

printf( “%d\n”, *p);    // prints 42
++p;                    // to what does p point now?
printf( “%d\n”, *p);    // what gets printed?
```

Adapted from Dennis Frey CMSC 313 Spring 2011

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 );
```

Adapted from Dennis Frey CMSC 313 Spring 2011

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 *`

Adapted from Dennis Frey CMSC 313 Spring 2011

sumArray

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

Adapted from Dennis Frey CMSC 313 Spring 2011

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