Pointers

Lecture 5
CS 113 – Spring 2008

Announcements

- Assignment 1 due today
  - 11:59PM on CMS
- We'll meet in 318 Phillips on Friday
  - Main goal: practice using pointers
  - Also: work on assignment, if you need time/help

Recall: C is call by value

- Modifying the value of a parameter in a function does not modify variables in the calling function

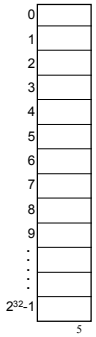
```
void foo(int x)
{
  x = 5;
}

int main(void)
{
  int y = 10;

  foo(y);
  printf("%d\n", y); // prints 10, not 5

  return 0;
}
```


Pointers

- What if a function needs to modify a parameter?
- Some languages let you pass variables by reference
  - As opposed to just passing the variable's value
- C has a more primitive solution: *pointers*
  - A pointer is just a memory address

Memory

- Variables are stored in memory
- Think of memory as a simple array
  - Every location in memory has an *address*
  - An address is an integer, just like an array index
- In C, a memory address is called a *pointer*
  - C lets you access memory locations directly



Declaring pointer variables

- A *pointer variable* stores a memory address
  - Use a * to declare a pointer variable
  - e.g.:

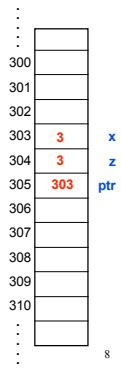
```
int *p1;   // declare a variable called p1 that stores a
           //  pointer to an integer

char *p2;  // declare a variable called p2 that stores a
           //  pointer to a character
```

Using pointers

- C has two unary operators related to pointers
  - `&` ("address of") operator
    - Returns the *address of* its argument
    - Said another way: returns *a pointer to* its argument
    - The argument must be a variable name
  - `*` ("dereference") operator
    - Returns the value stored at a given memory address
    - The argument must be a pointer

Pointers: a first example

```
int main( void )
{
  int x = 3, z;

  // create a pointer variable, and
  //  set it to the address of x
  int *ptr = &x;

  // set z to the value pointed to
  //  by ptr
  z = *ptr;

}
```

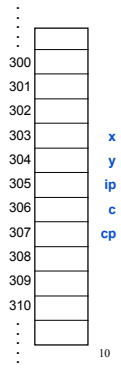


Don't get confused!

- Pointer notation is confusing, especially at first
  - `*`'s are used in two ways: in declarations, and as an operator
- In declarations, `*` creates a pointer variable, e.g.
  - `int *ptr;` declares a pointer to variables of type int
- As an operator, `*` dereferences a pointer, e.g.
  - `*ptr` returns the value stored in memory at the address contained in ptr

Another example

```
int main(void)
{
  int x = 1, y = 2;
  int *ip;
  char c;
  char *cp;

  ip = &x;
  printf( "%d\n", *ip );
  printf( "%d\n", *ip + 2 );
  y = *ip;
  *ip = 0;
  printf( "%d\n", x );
  cp = &x;
  *cp = 'z';
  cp = &c;
  *cp = 'z';
  printf( "%c\n", c );
  return 0;
}
```



Passing pointers to functions

- Recall that `foo` changes local variable `x`, but not the original variable `y`

```
void foo(int x)
{
  x = 5;
}

int main(void)
{
  int y = 10;

  foo(y);
  printf("%d\n", y); // prints 10, not 5

  return 0;
}
```


Passing pointers to functions

- This version does modify `y`:

```
void foo(int *x)
{
  *x = 5;
}

int main(void)
{
  int y = 10;

  foo( &y );
  printf("%d\n", y); // prints 5

  return 0;
}
```

Another example: swap

- This `swap()` function has no effect:

```
void swap ( int a, int b )
{
  int temp = a;
  a = b;
  b = temp;
}

void main() {
  int A = 1, B = 2;
  swap(A, B);
  printf( "%d %d\n", A, B );  // prints "1 2"
}
```


Another example: swap

- This `swap()` function works as expected

```
void swap ( int *a, int *b )
{
  int temp = *a;
  *a = *b;
  *b = temp;
}

void main() {
  int A = 1, B = 2;
  swap( &A, &B );
  printf( "%d %d\n", A, B );  // prints "2 1"
}
```


scanf

- scanf uses this technique to modify its parameters

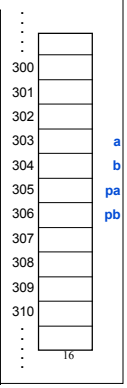
```
int I;
printf("enter an integer: ");
scanf("%d", &I);
```

passes pointer to `I` to scanf

- What happens if you forget the &?

Another example

```
void main() {
  int a = 3, b = 3;
  int *pa, *pb;

  pa = &a;
  pb = &b;
  if( pa == pb ) printf( "pa & pb equal.\n" );
  if( *pa == *pb ) printf( "*pa & *pb equal.\n" );

  (*pa)++;
  *pb += *pa;
  printf( "a: %d, b: %d\n", a, b );

  pb = pa;
  *pa += *pb;
  printf( "a: %d, b: %d\n", a, b );

  if( pa == pb ) printf( "pa & pb equal.\n" );
  if( *pa == *pb ) printf( "*pa & *pb equal.\n" );

  *((0 > 1) ? &a : &b) = 5;
}
```



Pointers are powerful and flexible

- Or: dangerous and hard to use
  - What does this code do?

```
void main() {
  char *x;
  *x = 'a';
}
```

  - What about this code?

```
void main() {
  char x='a', *p = &x;
  *p++;
  printf("%c\n", *p);
}
```


More dangers with pointers

- What does this code do?

```
int *function_3()
{
  int b;
  b = 3;
  return &b;
}

void main()
{
  int *a;
  a = function_3();
  printf( "a is equal to %d\n", *a );
}
```

Arrays

- To declare an array, use [ ], e.g.:

```
// create an array with 5 integer elements
int A[5];
```

  - Arrays in C are fixed size: their size can't be changed
  - The number between the brackets must be a constant

- You can give initial values for array elements, e.g.:

```
// create an array with 10 integer elements
int A[5] = {3, 7, -1, 4, 6};
```


Arrays

- Array indices in C are *zero-based*.
  - e.g. A[0], A[1], A[2], A[3], A[4]

- Example:

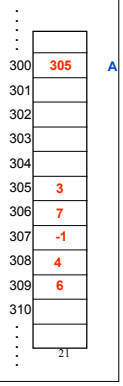
```
int main(void)
{
  int A[5] = {3, 7, -1, 4, 6};
  int j;
  double mean = 0;

  // compute mean of values in A
  for(j=0; j<5; j++)
    mean += A[j];

  mean /= 5;
  return 0;
}
```


Array and pointers

- Pointers and arrays are closely related
  - An array variable is actually just a pointer to the *first element in the array*

```
// create an array with 10 integer elements
int A[5] = {3, 7, -1, 4, 6};
```

- You can access array elements using array notation or pointers
  - A[0] is the same as *A
  - A[1] is the same as *(A+1)
  - A[2] is the same as *(A+2)
  - etc.



Arrays and pointers

- Accessing array elements using pointers:

```
int main(void)
{
  int A[5] = {3, 7, -1, 4, 6};
  int j;
  double mean = 0;

  // compute mean of values in A
  for(j=0; j<5; j++)
    mean += *(A+j);

  mean /= 5;
  return 0;
}
```