CMSC 313
Introduction to Computer Systems
Lecture 8
Pointers, cont.

Alan Sussman
als@cs.umd.edu

Administrivia

- Project 2 posted, due October 6
  - public tests posted
- Quiz on Wed. in discussion – up to pointers from today
- Read Reek, Chapter 6: Pointers

Chapter 6, Reek

POINTERS

Pointer example, again

- The **`*`** operator can be used on both the left and right sides of an assignment

```
int i = 6;
int j;
int *p;
p = &i;
j = *p;
printf("%d %d\n", i, j);
*p = 4;
printf("%d %d\n", i, j);
```

	Memory	Address
i	4	1084
j	6	1088
p	1084	1092

Garbage pointers

- When a pointer is declared, it points to whatever address was in the memory location allocated for the pointer (no initialization)
- Trying to dereference this random address will generally result in one of three Bad Things:
  – accessing a memory location you don't have permission to access (a "segmentation fault")
  – violating the computer's alignment policies (a "bus error")
  – silent failure: everything appears to work right... for now

`NULL` pointer

- This is a pointer that points to the address 0, where nothing is allowed to be accessed
- Defined in **`stddef.h`**, which is included by many other header files
- Analogue to Java's **`null`**
  – What happens when you try to call a method of an object which is **`null`**?
  – Very similar thing happens in C when trying to dereference a **`NULL`** pointer; it's usually a segfault
- Just like in Java, you have to check pointers to see if they're **`NULL`** **before** dereferencing them:

```
void f(int *p) {
  if (p != NULL)
    *p = 55;
}
```


Pointers to Pointers

- You can also obtain the address of a pointer variable:

```
int i = 4;
int j = 6;
int *p = &i;
int *q = &j;
int **r = &p;
printf("%d\n", **r);
*r = &j;
printf("%d\n", *p);
```

	Memory	Address
i	4	1084
j	6	1088
p	1088	1092
q	1088	1096
r	1092	1100

- This technique will be useful later when working with pointers as parameters

Pointers as parameters

- You can also pass addresses into a function:

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
...
int x = 2;
int y = 3;
swap(&x, &y);
printf("%d %d\n", x, y);
```

- Why do we need to use pointers here?

Structures and pointers

- We can also have pointers to structures:

```
typedef struct {
  int number, num_students, start_time;
} Section;

void add_students(Section *sec,
                  int students_to_add)
{
  (*sec).num_students += students_to_add;
}
...
Section s = {101, 25, 1300};
add_to_students(&s, 5);
printf("%d\n", s.num_students);
```


The **->** operator

- Dereferencing of a pointer to a structure must occur before accessing a field of the structure; due to precedence, parentheses are needed

```
Section s = {101, 25, 1300};
Section *sp = &s;
*sp.num_students += 5;   /* WRONG */
(*sp).num_students += 5; /* RIGHT */
```

- C has a special operator to make this easier:
  - "**(*sp).num_students**" is equivalent to "**sp->num_students**"

Don't forget to check for **NULL**

- A common error is to do something like this: assume **abs_val()** is supposed to return the absolute value of the number pointed to by **cp**, or return -1 if **cp** is **NULL**

```
typedef struct {
  double real;
  double imag;
} Complex;
...
double abs_val(Complex *cp) {
  double r = cp->real * cp->real;
  double i = cp->imag * cp->imag;
  if (cp == NULL)
    return -1;
  return r + i;
}
```

- Remember that the **->** is doing dereferencing; you must perform the **NULL** check before the pointer is dereferenced!

Generic pointers

- Pointers to **void** (**void ***) can point to any type:

```
void *vp;
int a, *ip;
double b, *dp;
vp = &a;
ip = vp;
vp = &b;
dp = vp;
vp = ip;
```

- No casts needed with **void *** pointers

Generic pointers, cont.

- You can't dereference a **`void *`** - you first need to cast or assign it to a real pointer type
  - the value obtained from a dereference depends on the type of pointer
- **`NULL`** is really defined as **`(void *) 0`**
- These allow use of generic code, but misuse can lead to the kinds of errors we've seen before:

```
void *vp;
int *ip;
double a = 3.14159;
vp = &a;
ip = vp;
printf("%d\n", *ip);  /* -266631570 */
```


Type conversion with pointers

- Converting from one type to a pointer has some uses:

```
unsigned int i;
unsigned char *ch;
i = 0x543210ab;
ch = (unsigned char *) &i;
printf("%d\n", *ch);
printf("%d\n",
       * (unsigned char *) &i);
```

- Prints out either MSB or LSB of **`i`**, depending on architecture

Type conversion, cont.

- Type conversion is very similar to what happens when we access an inactive union field

```
union {
  int i;
  double dbl;
} a;
double fp_val = 3.14159;
a.dbl = 3.14159;

printf("%d\n", a.i);  /* -266631570 */
printf("%d\n", * (int *) &fp_val);
```


Type conversion, cont.

- The two lines are the same!
- Although we stored a double-precision floating-point number at an address in memory, interpreting that address as a 4-byte integer results in a wildly different value from the floating-point number

10010101	01010110	10101010	11010100	00101010	01010100	10101010	10010011

The `const` modifier

- Indicates that a variable can't be changed, and enforced by compiler

```
int const i = 4;
const int j = 5;
i++;  /* ERROR */
j++;  /* ERROR */
```

- Order of type specifier and **`const`** modifier does matter when dealing with pointers:

```
int i = 4, j = 5;
const int *p = &i;  /* pointer to constant int */
int * const q = &j; /* constant pointer to int */
p = &j;    /* OK */
*p += 5;   /* ERROR */
q = &i;    /* ERROR */
*q += 23;  /* OK */
```

- The program **`cdecl`** can be useful for decoding some more complex declarations

Incrementing pointers

- Pointers can be incremented/decremented just like integer type variables, "moving" one element at a time
  - how much is added to the address depends on the size of the type to which the pointer points (as declared)
- Recall arrays are contiguous memory
- What does this function do?

```
int mystery(int array[]) {
  int *p = &(array[0]);
  int sum = 0;
  while (*p != -1) {
    sum += *p;
    p++;
  }
  return sum;
}
```

11	1148
22	1152
5	1156
-1	1160
2394	1164
45346	1168

Incrementing pointers, cont.

```
size_t strlen(const char *str) {
  size_t len = 0;
  while (*str) {
    len++;
    str++;
  }
  return len;
}
```

- Why can we move the **`str`** parameter?
- Why does this return the string's length?

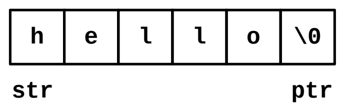
Incrementing pointers, cont.

- The postfix operators take precedence over the dereference operator and prefix operators
- **`*`** and prefix ops are at the same precedence level, and associate right to left
- **`++*p`** increments the value at the location to which **`p`** points, and evaluates to the incremented value
- **`*p++`** evaluates to the value at the location to which **`p`** points, and then advances **`p`**
- **`(*p)++`** evaluates to the value at the location to which **`p`** points, and then increments that value

Pointer arithmetic

- With two pointers in the same array, we can determine how far apart they are

```
size_t strlen(const char *str) {
  const char *ptr;
  for (ptr = str; *ptr; ptr++)
    ;
  return (size_t) (ptr – str);
}
```



Pointer arithmetic, cont.

- By adding an integer *n* to a pointer, we can get the address of the *n*th element past the element to which the pointer currently points

```
int arr[] = {2, 3, 5, 7, 11};
int *p = &(arr[0]);
int *q = p + 4;
printf("%d\n", *q);
```

Output: 11

- Only valid forms of pointer arithmetic:
  – pointer - pointer
  – pointer ± integer

Pointer arithmetic, cont.

- We can also use relational and equality operators when working with multiple pointers

```
void sum_subarray(int array[], int idx1, int idx2) {
  int *ptr;
  int sum = 0;
  ptr = array + idx1;
  while (ptr <= array + idx2) {
    sum += *ptr;
    ptr++;
  }
  return sum;
}
```