Lecture 3

Pointers

CPSC 275
Introduction to Computer Systems

Using **switch**

```
int aDigit = num % 16;
if (aDigit == 10)
   printf("A");
else if (aDigit == 11)
   printf("B");
else if (aDigit == 12)
   printf("C");
else if (aDigit == 13)
   printf("D");
else if (aDigit == 14)
   printf("E");
else if (aDigit == 15)
   printf("F");
else
   printf("%d", aDigit);
```

```
int aDigit = num % 16;
switch(aDigit) {
   case 10: printf("A");
            break;
   case 11: printf("B");
            break;
   case 12: printf("C");
            break;
   case 13: printf("D");
            break;
   case 14: printf("E");
            break;
   case 15: printf("F");
            break;
   default: printf("%d", aDigit);
}
```

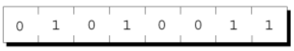
Using **switch**, cont'd

- Output when `aDigit = 15`?

```
int aDigit = num % 16;
switch(aDigit) {
   case 10: printf("A");
            break;
   case 11: printf("B");
            break;
   case 12: printf("C");
            break;
   case 13: printf("D");
            break;
   case 14: printf("E");
            break;
   case 15: printf("F");
            /* no break */
   default: printf("%d", aDigit);
}
```

Building Blocks of Memory

- The first step in understanding pointers is visualizing what they represent at the machine level.
- In most modern computers, main memory is divided into ***bytes,*** with each byte capable of storing eight bits of information:

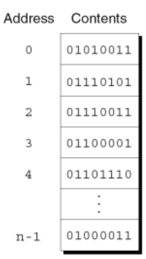


- Each byte has a unique ***address.***

4

Memory Address

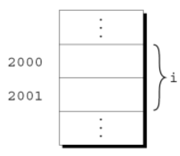
- If there are *n* bytes in memory, we can think of addresses as numbers that range from 0 to $n - 1$:



5

Variables in Memory

- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable `i` is 2000:



6

Pointer Variables

- Addresses can be stored in special ***pointer variables.***
- When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`.
- When a pointer variable is declared, its name must be preceded by an asterisk:

```
int *p;
```

- A graphical representation: 

Declaring Pointer Variables

- C requires that every pointer variable point only to a particular type (the ***referenced type***):

```
int *p;
double *q;
char *r;
```

- There are no restrictions on what the referenced type may be.

The Address and Indirection Operators

- C provides a pair of operators designed specifically for use with pointers.
  - To find the address of a variable, we use the **&** (*address*) operator.
  - To gain access to the object that a pointer points to, we use the ***** (*indirection*) operator.

The Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

```
int *p;  /* points nowhere in particular */
```

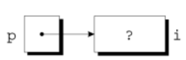
- It's crucial to initialize `p` before we use it.

The Address Operator

- One way to initialize a pointer variable is to assign it the address of a variable:

```
int i, *p;
…
p = &i;
```

- Assigning the address of `i` to the variable `p` makes `p` point to `i`:



The Indirection Operator

- Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object.
- If `p` points to `i`, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

The Indirection Operator

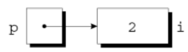
```
int i, *p;

p = &i;
```


```
i = 1;
```


```
printf("%d\n", i);
printf("%d\n", *p);

*p = 2;
```




```
printf("%d\n", i);
printf("%d\n", *p);
```


The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assigning a value to `*p` is particularly dangerous:

```
int *p;
*p = 1;    /*** WRONG ***/
```


Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type.
- Assume that the following declaration is in effect:

```
int i, j, *p, *q;
```

- Example of pointer assignment:

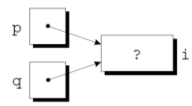
```
p = &i;
```


Pointer Assignment

- Another example of pointer assignment:

```
q = p;
```

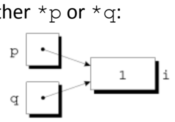
- `q` now points to the same place as `p`:



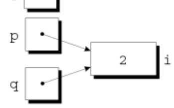
Pointer Assignment

- If `p` and `q` both point to `i`, we can change `i` by assigning a new value to either `*p` or `*q`:

```
*p = 1;
```




```
*q = 2;
```



- Any number of pointer variables may point to the same object.

Pointer Assignment

- Be careful not to confuse

```
q = p;
```

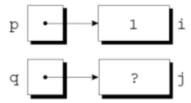
with

```
*q = *p;
```

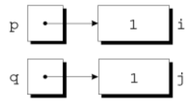
- The first statement is a pointer assignment, but the second is not.

Pointer Assignment

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



19

Pointers as Arguments

- Arguments in calls of `scanf` are pointers:

```
int i;
…
scanf("%d", &i);
```

Without the `&`, `scanf` would be supplied with the *value* of `i`.

20

The swap() function

- What's wrong with the following function?

```
int swap(int a, int b) // swap values of a and b
{
  int temp = a;
  a = b;
  b = temp;
}
```

- A correct version:

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

21

Pointers as Return Values

- Functions are allowed to return pointers:

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

- A call of the `max` function:

```
int *p, i, j;
…
p = max(&i, &j);
```

After the call, `p` points to either `i` or `j`.

22

Pointers as Return Values

- Never return a pointer to an *automatic* local variable:

```
int *f(void)
{
  int i;
  …
  return &i;
}
```

Why not?

The variable `i` won't exist after `f` returns.

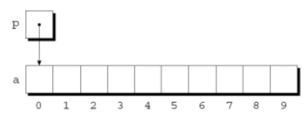
23

Pointer Arithmetic

- Pointer variables can point to array elements:

```
int a[10], *p;
p = &a[0];
```

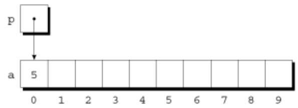
- A graphical representation:



24

Pointer Arithmetic

- We can now access `a[0]` through `p`; for example, we can store the value 5 in `a[0]` by writing
  `*p = 5;`
- An updated picture:



Pointer Arithmetic

- If `p` points to an element of an array `a`, the other elements of `a` can be accessed by performing ***pointer arithmetic*** (or ***address arithmetic***) on `p`.
- C supports three (and only three) forms of pointer arithmetic:
  - Adding an integer to a pointer
  - Subtracting an integer from a pointer
  - Subtracting one pointer from another

Adding an Integer to a Pointer

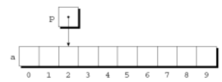
- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to.
- More precisely, if `p` points to the array element `a[i]`, then `p + j` points to `a[i+j]`.
- Assume that the following declarations are in effect:

```
int a[10], *p, *q, i;
```

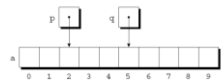

Adding an Integer to a Pointer

- Example of pointer addition:

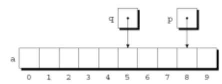
```
p = &a[2];
```



```
q = p + 3;
```



```
p += 6;
```

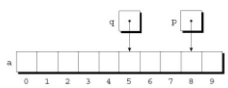


Subtracting an Integer from a Pointer

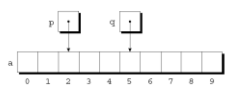
- If `p` points to `a[i]`, then `p - j` points to `a[i-j]`.
- Example:

```
p = &a[8];
```

```
q = p - 3;
```



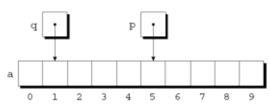
```
p -= 6;
```



Subtracting One Pointer from Another

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`.
- Example:

```
p = &a[5];
q = &a[1];
```



```
i = p - q;   /* i is 4 */
i = q - p;   /* i is -4 */
```

Comparing Pointers

- Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=).
- The outcome of the comparison depends on the relative positions of the two elements in the array.
- After the assignments

```
p = &a[5];
q = &a[1];
```

the value of `p` <= `q` ? 0

the value of `p` >= `q` ? 1

31