COMsW 1003-1

Introduction to Computer Programming in C

Lecture 11

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html
Announcements

• Grades for Homework 1 posted on Coursewors

• Homework 2 is due next Monday at the beginning of class

• Bring the printout to class!
Pointers
Pointers

Remember what happens when we declare a variable: the computer allocates memory for it.

```c
int x;
```

Value chosen by the computer

Address

033727FA88

Main memory

4 bytes (=32 bits)
Pointers

When we assign a value to a variable, the computer stores that value at the address in memory that was previously allocated for that variable.

```c
int x;
x = 3;
```

Address: 033727FA88

Main memory:

```
00000000 00000000 00000000 00000011
```

4 bytes (=32 bits)

```c
x *= 3; // x = 9
```

Main memory:

```
00000000 00000000 00000000 00001001
```
Pointers

Pointers are variables for memory addresses.
They are declared using the * operator.
They are called pointers because they **point to the place in memory** where other variables are stored.
How can we know what the address in memory of a variable is? The & operator.

```c
int x;
x = 3;

int *y;
y = &x;
```
Pointers - Syntax

When we declare a pointer, we must specify the type of variable it will be pointing to

```c
type *ptrName;
```

If we want to set a pointer to point to a variable, we must use the `&` operator

```c
ptrName = &varName;
```

```c
int x;
x = 3;

int *y;
y = &x;
```
Pointers : operators * and &

* dereference operator: gives the value in the memory pointed by a pointer (returns a value)

& reference operator: gives the address in memory of a variable (returns a pointer)

```c
int x = 3;
int *ptr;
ptr = &x;
*ptr = 5; // x = 5;
```

Main memory

00000000 00000000 00000000 00000011
Pointers : operators * and &

* dereference operator: gives the value in the memory pointed by a pointer (returns a value)

& reference operator: gives the address in memory of a variable (returns a pointer)

```
int x = 3;
int *ptr;
ptr = &x;
*ptr = 5;    // x = 5;
```

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Variable of type int</td>
</tr>
<tr>
<td>ptr</td>
<td>Pointer to an element of type int</td>
</tr>
<tr>
<td>&amp;x</td>
<td>Pointer to x</td>
</tr>
<tr>
<td>*ptr</td>
<td>Variable of type int</td>
</tr>
</tbody>
</table>
Pointers: operators * and &

* dereference operator: gives the value in the memory pointed by a pointer (returns a value)

& reference operator: gives the address in memory of a variable (returns a pointer)

```c
int x;
int *ptr;
```

- &x
- *ptr

- &ptr // pointer to a pointer
- *x // x is not a pointer
Pointers: operators * and &

* dereference operator: gives the value in the memory pointed by a pointer (returns a value)

& reference operator: gives the address in memory of a variable (returns a pointer)

```c
int x;
int *ptr;
&x
*ptr

This is weird but actually ok, we will see its meaning later
```

```c
&p
// pointer to a pointer
*x
// x is not a pointer
```
Pointers

Multiple pointers can point to the same address

```c
int x = 3, y = 2;
int *ptr = &x;
int *ptr2 = ptr;
```

```
Main memory
```

```
ptr
ptr2
```

```
0000 0000 0000 0011
0000 0000 0000 0010
```

NOTE: first 4 bits omitted to save space
Pointers

Multiple pointers can point to the same address

```c
int x = 3, y = 2;
int *ptr = &x;
int *ptr2 = ptr;
*ptr = 7;    // x = 7;
```

**Main memory**

```
0000 0000 0000 0111    // x
0000 0000 0000 0010    // y
```

NOTE: first 4 bits omitted to save space
Pointers

Multiple pointers can point to the same address

```c
int x = 3, y = 2;
int *ptr = &x;
int *ptr2 = ptr;

*ptr = 7;  // x = 7;
*ptr2 = *ptr2 + 1;  // x = 8;
```

![Memory Diagram](image)
Pointers

Multiple pointers can point to the same address

```c
int x = 3, y;

int *ptr = &x;

int *ptr2 = ptr;

*ptr = 7;   // x = 7;
*ptr2 = *ptr2 + 1;  // x = 8;

ptr = &y;

*ptr2 = 10; // x = 10;
```

Main memory

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>1010</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ptr2 is still pointing to x, even if ptr changed
Pointers

Be careful when using incremental operators!

```c
int x = 3;
int *ptr = &x;

*ptr++;    // x = ?
```

Main memory

0000 0000 0000 0011

In this case I am incrementing `ptr`, NOT the value of the variable pointed by it!
Pointers

Be careful when using incremental operators!

```
int x = 3;
int *ptr = &x;
(*ptr)++;  // x = 4;
```
Pointers and Arrays

• When set a pointer to an array, the pointer points to the **first element** in the array

```c
float arr[3] = {1, 2, 5};
float *pa;

pa = arr;
pa = &arr[0]; // These two notations are equivalent
```

• C automatically keeps pointer arithmetic in terms of the size of the variable type being pointed to

```c
arr[0] ↔ *pa
arr[1] ↔ *(pa+1)
```
Pointers and Arrays

- When set a pointer to an array, the pointer points to the **first element** in the array

```c
float arr[3] = {1, 2, 5};
float *pa;

pa = arr;
pa = &arr[0];
```

These two notations are equivalent

- C automatically keeps pointer arithmetic in terms of the size of the variable type being pointed to

```
arr[0] <-> *pa
arr[1] <-> *(pa+1)
```

Once we have set a pointer to the beginning of one array, we can use it as if it were the array itself!
Pointers and Arrays

When set a pointer to an array, the pointer points to the **first element** in the array.

```c
float arr[3] = {1, 2, 5};
float *p = arr;
*p = 5; // arr[0] = 5;
```
Pointers and Arrays

When set a pointer to an array, the pointer points to the **first element** in the array

```c
float arr[3] = {1, 2, 5};

float *p = arr;

*p = 5; // arr[0] = 5;
p++;

*p = 3; // arr[1] = 3;
```

![Memory diagram showing array elements and pointer movement]
Pointers and Arrays

When set a pointer to an array, the pointer points to the first element in the array

```c
float arr[3] = {1, 2, 5};
float *p = arr;
*p = 5; // arr[0] = 5;
p++;
*p = 3; // arr[1] = 3;
```

Note that for arrays, we do not need the reference & operator

Remember: an array is a set of elements of the same type allocated contiguously in memory!
Pointers and Arrays

```c
char *wPtrStart = word;
char *wPtrEnd = wPtrStart + strlen(word) - 1;

for( i=0 ; (i < strlen(word)/2) && (flag == 1) ; i++ ){

    if( *wPtrStart != *wPtrEnd ){
        flag = 0;
    }

    wPtrStart++;
    wPtrEnd--;
}
```
Pointers and Arrays

char *wPtrStart = word;
char *wPtrEnd = wPtrStart + strlen(word) - 1;

for (i = 0; (i < strlen(word)/2) && (flag == 1); i++){
    if ( *wPtrStart != *wPtrEnd ){
        flag = 0;
    }
    wPtrStart++;
    wPtrEnd--;
}

When we increment or decrement, the pointers move by 1 byte (pointers to char)
Pointers : operators * and &

Now we know exactly what happens in sscanf!

```c
sscanf( string, "format", &var1, ..., &varN);
```

Pointers to the addresses in memory where var1,..,varN are stored!
Functions
Passing arguments by value/reference

• **Pass by value** (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function

• **Pass by reference**: a pointer to the variable used at invocation time is passed to the function. We can modify the variable’s value inside the function
Passing arguments by value/reference

- **Pass by value** (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function.

```c
double computeCirc( double rad ){
    rad = 2;
    return(2 * rad * 3.14);
}
```

```c
int main(){
    double r = 5, circ;
    circ = computeCirc(r);
    return 0;
}
```
Functions
Passing arguments by value/reference

- **Pass by value** (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function

```c
double computeCirc(double rad) {
    rad = 2;
    return (2 * rad * 3.14);
}
```

- `r` is not affected by anything we do inside the function

```c
int main() {
    double r = 5, circ;
    circ = computeCirc(r);
    return 0;
}
```
Functions
Passing arguments by value/reference

• **Pass by reference**: a pointer to the variable used at invocation time is passed to the function. We can modify the variable’s value inside the function.

```c
double computeCirc( double *rad ){
    *rad = 2;
    return(2 * (*rad) * 3.14);
}
```

```c
int main(){
    double r = 5, circ;
    circ = computeCirc(&r);
    return 0;
}
```
Functions

Passing arguments by value/reference

- **Pass by reference**: a pointer to the variable used at invocation time is passed to the function. We can modify the variable’s value inside the function.

```c
double computeCirc( double *rad ){
    *rad = 2;
    return (2 * (*rad) * 3.14);
}

int main(){
    double r = 5, circ;
    circ = computeCirc(&r);
    return 0;
}
```

`r` has been modified!