CSE 220 – C Programming
Fall 2015

Pointers
Addresses in C
Address

- In C (and any other programming language), each variable corresponds to a "place" or "address" in the memory.

  - The address is an ID referring to specific location in memory assigned to the variable:
    - Just like your MSU PID
      - e.g., "A49550005" means you are the 49550005-th student who attends MSU
    - So the address/ID is just a very large number
      - e.g., your computer has 4GB memory, that is $4 \times 2^{30}$ bytes
      - each byte is assigned an ID, e.g., from 0 to $4 \times 2^{30}$
Address Operator &

- C allows two ways of accessing variables

  - Name
  - Address/Pointer
    - i.e., place in the memory the value(s) is stored
    - C allows you to "see" and operate on the address of that place

    & name_of_variable

Symbol & gets the address of the variable that follows it

[do NOT confuse with bitwise AND, unary versus binary]
Example

```c
#include <stdio.h>

int main( )
{
    int data = 100;
    float value = 56.47;
    printf("data is: %d\n", data);
    printf("value is: %f\n", value);
    return 0;
}
```

```
cse220:\>gcc mem.cc -o mem.exe
cse220:\>./mem.exe
data is: 100
value is: 56.47000
```

<table>
<thead>
<tr>
<th>value</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFF0</td>
<td>56.47</td>
</tr>
<tr>
<td>FFF1</td>
<td></td>
</tr>
<tr>
<td>FFF2</td>
<td></td>
</tr>
<tr>
<td>FFF3</td>
<td></td>
</tr>
<tr>
<td>FFF4</td>
<td>100</td>
</tr>
<tr>
<td>FFF5</td>
<td></td>
</tr>
<tr>
<td>FFF6</td>
<td></td>
</tr>
</tbody>
</table>
Example

The address operator &

Address of memory location allocated to each of the variables!

[address changes for each execution of]
Pointer Data Type in C
The **pointer data type**:

- A pointer is a variable whose value is the address of another variable

- Type: `Integer`

- Size: the number of bytes in which the target computer stores a memory address

- Provides **indirect** access to values
- **Pointer**: a variable used to store the **address** of another variable

- \( p \) and \( x \) both are **variables**, but:
  - \( p \) contains the (starting) address of \( x \) in the memory: \( p \) **points to** \( x \)
  - \( x \) contains a value
Declaration of Pointer Variables

Similar to other data types, you must declare a pointer before using it to store any variable address.

```
data_type *pointer_var_name;
```

- **data_type**: The type of data the pointer is defined for.
- **pointer_var_name**: The name of pointer variable.
- **Indirect operator**: Similar to other data types, you must declare a pointer before using it to store any variable address.
Declaration of Pointer Variables

data_type  *pointer_var_name;

- The pointer variable pointer_var_name:
  - is used to point to a value of type data_type
  - it is NOT storing the actual value of type data_type
  - is a variable storing an address

- The * before the pointer_var_name indicates that this is a pointer variable, not a regular variable

- The * is NOT a part of the pointer variable name

- By defining a pointer, memory is not allocated, unless you explicitly ask for it or assign existing addresses!
Examples of Pointer Variables

```
data_type  *pointer_var_name;
```

```
int  *ptr1;
float  *ptr2;
```

**ptr1** is a pointer to an **int** value i.e., it can have the **address of the memory location** (or the first of more than one memory locations) **allocated** to an **int** value

**ptr2** is a pointer to a **float** value i.e., it can have the **address of the memory location** (or the first of more than one memory locations) **allocated** to a **float** value
Declaration of Pointer Variables

Whitespace doesn’t matter and each of the following will declare `ptr` as a pointer (to a `float`) variable and `data` as a `float` variable

```c
float *ptr, data;
float* ptr, data;
float (*ptr), data;
float data, *ptr;
```

Pointer variables can appear with other declarations:

```c
float *ptr, data, arr[20];
```
Why type (I) ?

Note that, a pointer (or an address) has its own "data type"
- You can view
  ```
  int *ptr;
  ```
in this way:
  ```
  int* ptr;
  ```
- i.e, `int*` is a type (we may call it the "integer pointer type"), which means the value stored in variable `ptr` is an address of an integer value

**Question**: Why do we need to specify the type of pointer?
Why type (II)?

- **Question**: Why do we need to specify the type of pointer?

Remember that the address of a value is just the address of the **starting/first byte**, that's why we need to know the type, i.e., **how many bytes the value occupies**.

**Question 1**: how many bytes should I read?

**Question 2**: what data type I am supposed to convert 0 and 1's?
How to Use Pointers?

1. How to assign the address of a variable to a pointer?

2. How access the value at the address available in the pointer variable?
How to Use Pointers?

1. How to assign the address of a variable to a pointer?

```c
int data = 10;
int *ptr;

ptr = &data;
```

2. How access the value at the address available in the pointer variable?
How to Use Pointers?

1. How to assign the address of a variable to a pointer?

```c
int data = 10;
int *ptr;

ptr = &data;
```

2. How access the value at the address available in the pointer variable? [indirect operator]

```c
int data = 10;
int *ptr;
ptr = &data;

*ptr = 20;
```
Recap: Twin Operators

1. Define pointers by:

   ```
   data_type  *pointer_var_name;
   ```

2. From a variable to its address:

   Address Operator: `& + (variable)`

3. From an address to its content:

   Indirect Operator: `* + (address)`

   `*ptr`: the content of memory location ptr refers to
Caution

float data, *ptr;

ptr does not point to any particular place
Must initialize ptr before using it

ptr = &data;

Assign address of data to ptr using address operator &
ptr is a pointer to data

float data, *ptr = &data; [data must be declared first]

Combine declaration and initialization
Twin Operators:
Address Operator (&) + Indirection Operator (*)

float data = 50.8;
float *ptr;
ptr = &data;

data

<table>
<thead>
<tr>
<th>FFF0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FFF1</td>
<td></td>
</tr>
<tr>
<td>FFF2</td>
<td></td>
</tr>
<tr>
<td>FFF3</td>
<td></td>
</tr>
<tr>
<td>FFF4</td>
<td>50.8</td>
</tr>
<tr>
<td>FFF5</td>
<td></td>
</tr>
<tr>
<td>FFF6</td>
<td></td>
</tr>
</tbody>
</table>
float data = 50.8;
float *ptr;
ptr = &data;

Twin Operators:
Address Operator (&) + Indirection Operator (*)
Twin Operators: Address Operator (&) + Indirection Operator (*)

float data = 50.8;
float *ptr;
ptr = &data;
Indirection Operator: *

- So, we have two ways to access (read or write) a "place" in the memory
  
  - **Directly**, use the variable which is "assigned" to that place, or in other words, we give that place a “name"

```c
int a;   // a can be viewed as the "name" for a place
a = 99;
printf("%d", a);
```
• **Indirectly**, use the address of that place, and the address is stored in another variable, i.e., the pointer variable (that is why we call * as "Indirection Operator")

```c
int a;

int *p = &a;

*p = 99;
*p++
*p = *p - 100
printf("%d", *p);
```
Indirection Operator: *

```c
int a;
int *p = &a;
*p = 99;
*p++
*p = *p - 100
printf("%d", *p);
```

- We can just treat \*p as a variable, which is equivalent to a
  - \*p is also called the "alias" of a
  - when we use a, we can also use \*p, as long as p still points to a
The indirection operator * is used to access the value pointed to by a pointer:

i.e., the value of \( p \) is an address, and \( *p \) is the value stored in the place where that address points to.

```
int i = 11;
int *p = &i;
printf("\%d\n", *p);
printf("\%x\n", p);
```
The indirection operator * is used to access the value pointed to by a pointer:

i.e., the value of \( p \) is an address, and \( *p \) is the value stored in the place where that address points to

\[
\begin{array}{c|c|c}
\text{i} & \text{0x7fffffffda6c} & 11 \\
\text{p} & \text{0x7fffffffda70} & \text{0x7fffffffda6c}
\end{array}
\]

```c
int i = 11;
int *p = &i;
printf("%d\n", *p); // prints 11
printf("%x\n", p); // 0x7fffffffda6c
```
Example (I)

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

- **&i**: address of `i`
- `*(&i)`: value pointed to by the pointer

```
&i: address of i
*(&i): value pointed to by the pointer

*i &i same as i
```
Example (II)

```c
int i, j, *p;
i = 3;

j = *(&i);
p = &i;

i = 4;

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

```c
int i, j, *p;
i = 3;
j = *(&i);
p = &i;
i = 4;
printf(%d
", i);
printf(%d
", j);
printf(%d
", *p);
```

Output: 4, 3, 4 [p points to i, so it prints the current value of i]
Example

1: int i = 3, *p = &i;
2: *p = 4;
3: printf(%d
", i);
4: printf(%d
", *p);

Line 2: Assigns 4 to the value pointed to by p, so assigns 4 to i
Example

1: int i = 3, *p = &i;
2: *p = 4;
3: printf(%d
”, i);
4: printf(%d
”, *p);

Line 2: Assigns 4 to the value pointed to by p, so assigns 4 to i
Line 3: Prints 4
Line 4: Prints 4
Dereferencing Example

```c
#include <stdio.h>

int main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    printf("ptr: %f\n", *ptr);
    *ptr = 27.4;
    printf("ptr: %f\n", *ptr);
    printf("ptr: %f\n", data);
    return 0;
}
```

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFF0</td>
<td></td>
</tr>
<tr>
<td>FFF1</td>
<td></td>
</tr>
<tr>
<td>FFF2</td>
<td></td>
</tr>
<tr>
<td>FFF3</td>
<td></td>
</tr>
<tr>
<td>FFF4</td>
<td>50.8</td>
</tr>
<tr>
<td>FFF5</td>
<td></td>
</tr>
<tr>
<td>FFF6</td>
<td></td>
</tr>
</tbody>
</table>
Dereferencing Example

```
#include <stdio.h>

int main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    *ptr = 27.4;
    printf("ptr: %f\n", *ptr);
    printf("ptr: %f\n", ptr);
    printf("ptr: %f\n", data);
    return 0;
}
```
Dereferencing Example

```c
#include <stdio.h>

int main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    printf("ptr: %f\n", *ptr);
    *ptr = 27.4;
    printf("ptr: %f\n", *ptr);
    printf("ptr: %f\n", data);
    return 0;
}
```

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFF0</td>
<td></td>
</tr>
<tr>
<td>FFF1</td>
<td></td>
</tr>
<tr>
<td>FFF2</td>
<td></td>
</tr>
<tr>
<td>FFF3</td>
<td></td>
</tr>
<tr>
<td>FFF4</td>
<td>27.4</td>
</tr>
<tr>
<td>FFF5</td>
<td></td>
</tr>
<tr>
<td>FFF6</td>
<td></td>
</tr>
</tbody>
</table>
Dereferencing Example

```c
#include <stdio.h>

int main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    printf("ptr: %f\n", *ptr);
    *ptr = 27.4;
    printf("ptr: %f\n", *ptr);
    printf("ptr: %f\n", data);
    return 0;
}
```
Dereferencing Example

```c
#include <stdio.h>

int main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    printf("ptr: %f\n", *ptr);
    *ptr = 27.4;
    printf("ptr: %f\n", *ptr);
    printf("ptr: %f\n", data);
    return 0;
}
```

Table:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFF0</td>
<td>FFF4</td>
</tr>
<tr>
<td>FFF1</td>
<td></td>
</tr>
<tr>
<td>FFF2</td>
<td></td>
</tr>
<tr>
<td>FFF3</td>
<td></td>
</tr>
<tr>
<td>FFF4</td>
<td>27.4</td>
</tr>
<tr>
<td>FFF5</td>
<td></td>
</tr>
<tr>
<td>FFF6</td>
<td></td>
</tr>
</tbody>
</table>
Dereferencing Example

```c
#include <stdio.h>

int main()
{
  float data = 50.8;
  float *ptr;
  ptr = &data;
  printf("ptr: %f\n", *ptr);
  *ptr = 27.4;
  printf("ptr: %f\n", *ptr);
  printf("ptr: %f\n", data);
  return 0;
}
```

[cse220:]>gcc pointer.cc -o pointer.exe
[cse220:]>./pointer.exe
ptr: 50.799999
ptr: 27.400000
ptr: 27.400000
[cse220:]>
int a, *p, *q;

p = &a;  //copy the address of a into p
q = p;   //copy the content of p into q
*p = 5;       //change the value pointed to by p to 5
printf("%d\n", *q);  // output?

---

Diagram:

```
p
   5
q
   a
```
int a = 5, *p = &a;
int b = 7, *q = &b;

q = p;

Copies the content of p to the content of q
int a = 5, *p = &a;
int b = 7, *q = &b;

*p = *p;

Copies value pointed to by p to value pointed to by q
The **NULL** pointer

A pointer can be initialized during declaration by assigning it the address of an existing variable

```c
float data = 50.8;
float *ptr = &data;
```

If a pointer is not initialized during declaration, it is wise to give it a **NULL** (0) value

```c
int *ip = 0;
float *fp = NULL;
```
The **NULL** pointer is a valid address for any data type. But **NULL** is not memory address 0.

It is an error to dereference a pointer whose value is **NULL**.
- Such an error may cause your program to crash, or behave erratically.
- It is the programmer’s job to check for this.