Pointers: Basics
What is a pointer?

- First of all, it is a variable, just like other variables you studied
  - So it has type, storage etc.
- **Difference:** it can only store the address (rather than the value) of a data item
- Type of a pointer variable – pointer to the type of the data whose address it will store
  - Example: int pointer, float pointer,…
  - Can be pointer to any user-defined types also like structure types
They have a number of useful applications
   □ Enables us to access a variable that is defined outside the function
   □ Can be used to pass information back and forth between a function and its reference point
   □ More efficient in handling data tables
   □ Reduces the length and complexity of a program
   □ Sometimes also increases the execution speed
Basic Concept

- As seen before, in memory, every stored data item occupies one or more contiguous memory cells
  - The number of memory cells required to store a data item depends on its type (char, int, double, etc.).
- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.
  - Since every byte in memory has a unique address, this location will also have its own (unique) address.
Consider the statement

\[ \text{int } \text{xyz} = 50; \]

- This statement instructs the compiler to allocate a location for the integer variable \text{xyz}, and put the value 50 in that location.
- Suppose that the address location chosen is 1380

\begin{tabular}{|c|c|}
\hline
\text{xyz} & variable \\
50 & value \\
1380 & address \\
\hline
\end{tabular}
Contd.

- During execution of the program, the system always associates the name \textit{xyz} with the address \texttt{1380}
  - The value \texttt{50} can be accessed by using either the name \textit{xyz} or the address \texttt{1380}

- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory
  - Such variables that hold memory addresses are called \texttt{pointers}
  - Since a pointer is a variable, its value is also stored in some memory location
Contd.

- Suppose we assign the address of *xyz* to a variable *p*
  - *p* is said to point to the variable *xyz*

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>xyz</em></td>
<td>50</td>
<td>1380</td>
</tr>
<tr>
<td><em>p</em></td>
<td>1380</td>
<td>2545</td>
</tr>
</tbody>
</table>

p = &xyz;
Address vs. Value

- Each memory cell has an address associated with it
## Address vs. Value

- Each memory cell has an address associated with it.
- Each cell also stores some value.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Address vs. Value

- Each memory cell has an **address** associated with it
- Each cell also stores some **value**
- Don’t confuse the **address** referring to a memory location with the **value** stored in that location

101 102 103 104 105 ...

... 23 ... 42 ...
Values vs Locations

- Variables name memory locations, which hold values

```
1024: 32 -> value
     x -> name
     address
```
Pointers

- A pointer is just a C variable whose value can contain the address of another variable.
- Needs to be declared before use just like any other variable.
- General form:

  ```c
  data_type *pointer_name;
  ```

- Three things are specified in the above declaration:
  - The asterisk (*) tells that the variable `pointer_name` is a pointer variable.
  - `pointer_name` needs a memory location.
  - `pointer_name` points to a variable of type `data_type`.
Example

```c
int  *count;
float  *speed;
char *c;
```

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like:

```c
int *p, xyz;

p = &xyz;
```

This is called pointer initialization.
Pointers can be defined for any type, including user defined types

Example

```
struct name {
    char first[20];
    char last[20];
};
struct name *p;
```

- p is a pointer which can store the address of a `struct name` type variable
Accessing the Address of a Variable

- The address of a variable is given by the & operator
  - The operator & immediately preceding a variable returns the address of the variable

- Example:
  
  \[
  p = &xyz;
  \]
  - The address of \( xyz \) (1380) is assigned to \( p \)

- The & operator can be used only with a simple variable (of any type, including user-defined types) or an array element
  
  \[
  \&distance
  \]
  \[
  \&x[0]
  \]
  \[
  \&x[i-2]
  \]
Illegal Use of &

- &235
  - Pointing at constant

- int arr[20];
  - &arr;
    - Pointing at array name

- &(a+b)
  - Pointing at expression

In all these cases, there is no storage, so no address either
#include <stdio.h>
int main()
{
    int a;
    float b, c;
    double d;
    char ch;

    a = 10;    b = 2.5;    c = 12.36;    d = 12345.66;    ch = 'A';
    printf ("%d is stored in location %u \n", a, &a); 
    printf ("%f is stored in location %u \n", b, &b); 
    printf ("%f is stored in location %u \n", c, &c); 
    printf ("%lf is stored in location %u \n", d, &d); 
    printf ("%c is stored in location %u \n", ch, &ch); 
    return 0;
}
Output

10 is stored in location 3221224908
2.500000 is stored in location 3221224904
12.360000 is stored in location 3221224900
12345.660000 is stored in location 3221224892
A is stored in location 3221224891
Accessing a Variable Through its Pointer

- Once a pointer has been assigned the address of a variable, the value of the variable can be accessed using the indirection operator (*).

```c
int a, b;
int *p;
p = &a;
b = *p;
```

Equivalent to

```
b = a;
```
Example

```c
#include <stdio.h>
int main()
{
    int   a, b;
    int   c = 5;
    int   *p;

    a  =  4  *  (c  +  5) ;
    p  =  &c;
    b  =  4  *  (*p  +  5) ;
    printf ("a=%d  b=%d \n",  a, b);
    return 0;
}
```

Equivalent

```
a=40  b=40
```
Example

```c
int main()
{
    int x, y;
    int *ptr;

    x = 10;
    ptr = &x;
    y = *ptr;

    printf ("%d is stored in location %u \n", x, &x);
    printf ("%d is stored in location %u \n", *ptr, &x);
    printf ("%d is stored in location %u \n", *ptr, ptr);
    printf ("%d is stored in location %u \n", y, &x);
    printf ("%u is stored in location %u \n", ptr, &ptr);
    printf ("%d is stored in location %u \n", y, &y);

    *ptr = 25;
    printf ("\n\Now x = %d \n", x);

    return 0;
}
```
Suppose that

<table>
<thead>
<tr>
<th>Address of x:</th>
<th>3221224908</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of y:</td>
<td>3221224904</td>
</tr>
<tr>
<td>Address of ptr:</td>
<td>3221224900</td>
</tr>
</tbody>
</table>

Then output is

10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
3221224908 is stored in location 3221224900
10 is stored in location 3221224904

Now x = 25
Example

```c
int x;
int *xp;

xp = &x;

*xp = 0; /* Assign 0 to x */
*xp = *xp + 1; /* Add 1 to x */
```
Value of the pointer

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
  - Local variables in C are not initialized, they may contain anything

- After declaring a pointer:
  ```
  int *ptr;
  ```
  `ptr` doesn’t actually point to anything yet. We can either:
  - make it point to something that already exists, or
  - allocate room in memory for something new that it will point to… (dynamic allocation, to be done later)
Example

Memory and Pointers:
Memory and Pointers:

```c
int *p, v;
```

![Diagram showing memory allocation for variables `p` and `v` with arbitrary values at addresses 2300 and 1500, respectively.]
Memory and Pointers:

```c
int v, *p;
p = &v;
```

Diagram:

- Variable `v`:
  - Address: 1500
  - Value: arbitrary value

- Pointer `p`:
  - Address: 1500
  - Value: 2300
Memory and Pointers:

```c
int v, *p;
p = &v;
v = 17;
```
Memory and Pointers:

```c
int v, *p;
p = &v;
v = 17;
*p = *p + 4;
v = *p + 4
```
More Examples of Using Pointers in Expressions

If p1 and p2 are two pointers, the following statements are valid:

\[
\begin{align*}
\text{sum} &= \ast p1 + \ast p2; \\
\text{prod} &= \ast p1 \ast \ast p2; \\
\text{prod} &= (\ast p1) \ast (\ast p2); \\
\ast p1 &= \ast p1 + 2; \\
x &= \ast p1 / \ast p2 + 5;
\end{align*}
\]

Note that this \textit{unary} \* has higher precedence than all arithmetic/relational/logical operators.
Things to Remember

- Pointer variables must always point to a data item of the same type
  ```
  float x;
  int *p;
  p = &x;
  ```
  will result in wrong output
- Never assign an absolute address to a pointer variable
  ```
  int *count;
  count = 1268;
  ```
Pointer Expressions

- Like other variables, pointer variables can appear in expressions

- What are allowed in C?
  - Add an integer to a pointer
  - Subtract an integer from a pointer
  - Subtract one pointer from another (related)

- If $p_1$ and $p_2$ are both pointers to the same array, then $p_2 - p_1$ gives the number of elements between $p_1$ and $p_2$
Contd.

- What are not allowed?
  - Adding two pointers.
    \[ p1 = p1 + p2; \]
  - Multiply / divide a pointer in an expression
    \[ p1 = p2 / 5; \]
    \[ p1 = p1 - p2 * 10; \]
Scale Factor

- We have seen that an integer value can be added to or subtracted from a pointer variable.

```c
int *p1, *p2;
int i, j;

p1 = p1 + 1;
p2 = p1 + j;
p2++;  
p2 = p2 - (i + j);
```

- In reality, it is not the integer value which is added/subtracted, but rather the scale factor times the value.
## Data Type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Scale Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

- If `p1` is an integer pointer, then `p1++` will increment the value of `p1` by 4
The scale factor indicates the number of bytes used to store a value of that type

- So the address of the next element of that type can only be at the (current pointer value + size of data)

The exact scale factor may vary from one machine to another

Can be found out using the `sizeof` function

- Gives the size of that data type

Syntax:

```
sizeof (data_type)
```
int main()
{
    printf("No. of bytes in int is %u \n", sizeof(int));
    printf("No. of bytes in float is %u \n", sizeof(float));
    printf("No. of bytes in double is %u \n", sizeof(double));
    printf("No. of bytes in char is %u \n", sizeof(char));
    printf("No. of bytes in int * is %u \n", sizeof(int *));
    printf("No. of bytes in float * is %u \n", sizeof(float *));
    printf("No. of bytes in double * is %u \n", sizeof(double *));
    printf("No. of bytes in char * is %u \n", sizeof(char *));
    return 0;
}
- Note that pointer takes 4 bytes to store, independent of the type it points to
- However, this can vary between machines
  - Output of the same program on a server

<table>
<thead>
<tr>
<th>Type</th>
<th>No. of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>int *</td>
<td>8</td>
</tr>
<tr>
<td>float *</td>
<td>8</td>
</tr>
<tr>
<td>double *</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>8</td>
</tr>
</tbody>
</table>

- Always use `sizeof()` to get the correct size
- Should also print pointers using `%p` (instead of `%u` as we have used so far for easy comparison)
int main()
{
    int A[5], i;

    printf(“The addresses of the array elements are:\n”);
    for (i=0; i<5; i++)
        printf(“&A[%d]: Using %p = %p, Using %u = %u”, i, &A[i], &A[i]);
    return 0;
}

Output on a server machine

&A[0]: Using %p = 0x7fffb2ad5930, Using %u = 2997705008
&A[1]: Using %p = 0x7fffb2ad5934, Using %u = 2997705012
&A[2]: Using %p = 0x7fffb2ad5938, Using %u = 2997705016
&A[3]: Using %p = 0x7fffb2ad593c, Using %u = 2997705020
&A[4]: Using %p = 0x7fffb2ad5940, Using %u = 2997705024

0x7fffb2ad5930 = 140736191093040 in decimal (NOT 2997705008)
so print with %u prints a wrong value (4 bytes of unsigned int cannot hold 8 bytes for the pointer value)