Pointers

- Address-Of Operator &
- Pointer Variables
- Indirection (Dereference) Operator *
- The NULL Pointer
- new
- delete
- an Inaccessible Object
- a Dangling Pointer
Introduction To Pointers and Dynamic Variables

• pointer
  – stores a memory address
  – a positive integer or 0
  – 0 is special, a NULL address ("points" to nothing)

• range of values
  – depends upon the computer you use
  – 0 up through some maximum
Legal Operations && Uses

• Legal Operations
  – assignment
  – comparison ( == , < , > etc. )
  – arithmetic: addition and subtraction only

• Uses
  – access a memory location via a pointer
  • reference parameters
  – set up dynamic variables and data structures
Addresses in Memory

- when a variable is declared,
  - enough memory to hold a value of that type is allocated for it at an unused memory location.
  - This is the address of the variable

```plaintext
int x;
float number;
char ch;
```

![Diagram showing memory allocation for x, number, and ch]
Obtaining Memory Addresses

- The address of a non-array variable can be obtained by using the `address-of` operator

```cpp
int x;
float number;
char ch;

cout << "Address of x is " << &x << endl;
cout << "Address of number is " << &number << endl;
cout << "Address of ch is " << &ch << endl;
```
What is a pointer variable?

• a **variable** whose value is the **address** of a location in memory.

• to declare a pointer variable,
  – must specify type of value that the pointer will point to

```c
int* ptr; // ptr will hold the memory address
// of an int

char* q; // q will hold the memory address
// of a char
```
Using a Pointer Variable

```cpp
int  x;
x  = 12;

int*  ptr;
ptr  = &x;
cout  <<  ptr;

Output is: 2000
```
Unary operator *

```cpp
int x;
x = 12;

int* ptr;
ptr = &x;

cout << *ptr;
```

Output is: **12**

**NOTE:** 
*ptr
The value pointed to by ptr

* is the **indirection (dereference)** operator
Using the Dereference Operator

```c
int x;
x = 12;

int* ptr;
ptr = &x;
*ptr = 5;
```
Using the Dereference Operator

```c
int x;
x = 12;

int* ptr;
ptr = &x;

*ptr = 5;
// changes the value
// at address ptr to 5
```

What if I did:
```
ptr = 5;
```
char ch;
ch = 'A';

char* q;
q = &ch;

*q = 'Z';
char* p;
p = q;

// now p and q both point to ch
What’s in memory?

```c
int* p;
int* q;
int num, val;
```

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>1001</td>
<td>?</td>
</tr>
<tr>
<td>1002</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>1003</td>
<td>?</td>
</tr>
<tr>
<td>1004</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>num</td>
<td>1005</td>
<td>?</td>
</tr>
<tr>
<td>1006</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>val</td>
<td>1007</td>
<td>?</td>
</tr>
</tbody>
</table>
What’s in memory?

```c
int* p;
int* q;
int num, val;

num = 5;
p = &num;
```

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>1001</td>
<td>?</td>
</tr>
<tr>
<td>q</td>
<td>1002</td>
<td>?</td>
</tr>
<tr>
<td>num</td>
<td>1003</td>
<td>?</td>
</tr>
<tr>
<td>num</td>
<td>1004</td>
<td>?</td>
</tr>
<tr>
<td>num</td>
<td>1005</td>
<td>?</td>
</tr>
<tr>
<td>val</td>
<td>1006</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>1007</td>
<td>?</td>
</tr>
</tbody>
</table>
What’s in memory?

```c
int* p;
int* q;
int num, val;

num = 5;
p = &num;
```

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>1001</td>
<td>1005</td>
</tr>
<tr>
<td>q</td>
<td>1003</td>
<td>?</td>
</tr>
<tr>
<td>num</td>
<td>1005</td>
<td>5</td>
</tr>
<tr>
<td>val</td>
<td>1006</td>
<td>?</td>
</tr>
</tbody>
</table>
int* p;
int* q;
int num, val;
num = 5;
p = &num;
val = *p;

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>1001</td>
<td>1005</td>
</tr>
<tr>
<td>q</td>
<td>1003</td>
<td>?</td>
</tr>
<tr>
<td>num</td>
<td>1004</td>
<td>?</td>
</tr>
<tr>
<td>val</td>
<td>1006</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>1007</td>
<td>?</td>
</tr>
</tbody>
</table>
What’s in memory?

```c
int* p;
int* q;
int num, val;
num = 5;
p = &num;
val = *p;
```

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>1001</td>
<td>1005</td>
</tr>
<tr>
<td></td>
<td>1002</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1003</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>1004</td>
<td>?</td>
</tr>
<tr>
<td>num</td>
<td>1005</td>
<td>5</td>
</tr>
<tr>
<td>val</td>
<td>1006</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1007</td>
<td>?</td>
</tr>
</tbody>
</table>
What’s in memory?

```plaintext
int* p;
int* q;
int num, val;
num = 5;
p = &num;
val = *p;
val = *p + 7;
q = p;
```

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>1001</td>
<td>1005</td>
</tr>
<tr>
<td>q</td>
<td>1002</td>
<td>?</td>
</tr>
<tr>
<td>num</td>
<td>1003</td>
<td>?</td>
</tr>
<tr>
<td>num</td>
<td>1004</td>
<td>?</td>
</tr>
<tr>
<td>val</td>
<td>1005</td>
<td>5</td>
</tr>
<tr>
<td>val</td>
<td>1006</td>
<td>5</td>
</tr>
<tr>
<td>q</td>
<td>1007</td>
<td>?</td>
</tr>
</tbody>
</table>
What’s in memory?

```c
int* p;
int* q;
int num, val;
num = 5;
p = &num;
val = *p;
val = *p + 7;
q = p;
```

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>1001</td>
<td>1005</td>
</tr>
<tr>
<td>num</td>
<td>1002</td>
<td>?</td>
</tr>
<tr>
<td>q</td>
<td>1003</td>
<td>1005</td>
</tr>
<tr>
<td>num</td>
<td>1004</td>
<td>?</td>
</tr>
<tr>
<td>val</td>
<td>1005</td>
<td>5</td>
</tr>
<tr>
<td>val</td>
<td>1006</td>
<td>12</td>
</tr>
<tr>
<td>q</td>
<td>1007</td>
<td>?</td>
</tr>
</tbody>
</table>
Array and Pointers

- const int SIZE = 5;
  int list [SIZE] = {3, 5, 7, 8, 9};
- list
  - is a pointer constant
  - value is
    - memory address of the 1st element in array list
    - list == &list[0]
    - (determined by compiler)
int list[size] = {3, 5, 7, 8, 9}

p = list

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>3003</td>
<td>2500</td>
</tr>
<tr>
<td>list[0]</td>
<td>2500</td>
<td>3</td>
</tr>
<tr>
<td>list[1]</td>
<td>2501</td>
<td>5</td>
</tr>
<tr>
<td>list[2]</td>
<td>2502</td>
<td>7</td>
</tr>
<tr>
<td>list[3]</td>
<td>2503</td>
<td>8</td>
</tr>
<tr>
<td>list[4]</td>
<td>2504</td>
<td>9</td>
</tr>
</tbody>
</table>
```plaintext
Possible Memory Map

int list[size] = {3, 5, 7, 8, 9}
p = list

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>3003</td>
<td>2500</td>
</tr>
<tr>
<td>list[0]</td>
<td>2500</td>
<td>3</td>
</tr>
<tr>
<td>list[1]</td>
<td>2501</td>
<td>5</td>
</tr>
<tr>
<td>list[2]</td>
<td>2502</td>
<td>7</td>
</tr>
<tr>
<td>list[3]</td>
<td>2503</td>
<td>8</td>
</tr>
<tr>
<td>list[4]</td>
<td>2504</td>
<td>9</td>
</tr>
</tbody>
</table>
```

p = list + 2; cout << *p
prints 2500
```plaintext
int list[size] = {3, 5, 7, 8, 9}
p = list
p = list + 2;
```

### Possible Memory Map

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>3003</td>
<td>2500</td>
</tr>
<tr>
<td>list[0]</td>
<td>2500</td>
<td>3</td>
</tr>
<tr>
<td>list[1]</td>
<td>2501</td>
<td>5</td>
</tr>
<tr>
<td>list[2]</td>
<td>2502</td>
<td>7</td>
</tr>
<tr>
<td>list[3]</td>
<td>2503</td>
<td>8</td>
</tr>
<tr>
<td>list[4]</td>
<td>2504</td>
<td>9</td>
</tr>
</tbody>
</table>
Possible Memory Map

int list[size] = {3, 5, 7, 8, 9}

p = list

p = list + 2;

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>3003</td>
<td>2500</td>
</tr>
<tr>
<td>list[0]</td>
<td>2500</td>
<td>3</td>
</tr>
<tr>
<td>list[1]</td>
<td>2501</td>
<td>5</td>
</tr>
<tr>
<td>list[2]</td>
<td>2502</td>
<td>7</td>
</tr>
<tr>
<td>list[3]</td>
<td>2503</td>
<td>8</td>
</tr>
<tr>
<td>list[4]</td>
<td>2504</td>
<td>9</td>
</tr>
</tbody>
</table>
Possible Memory Map

int list[size] = {3, 5, 7, 8, 9}

p = list

p = list + 2;

cout << *p

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>3003</td>
<td>2500</td>
</tr>
<tr>
<td>list[0]</td>
<td>2500</td>
<td>3</td>
</tr>
<tr>
<td>list[1]</td>
<td>2501</td>
<td>5</td>
</tr>
<tr>
<td>list[2]</td>
<td>2502</td>
<td>7</td>
</tr>
<tr>
<td>list[3]</td>
<td>2503</td>
<td>8</td>
</tr>
<tr>
<td>list[4]</td>
<td>2504</td>
<td>9</td>
</tr>
</tbody>
</table>
int list[size] = {3, 5, 7, 8, 9}

p = list

p = list + 2;

cout << *p

prints 7
C strings

```c
char str[8];
```

- `str` is the **base address** of the array.
- `str` is a pointer
  - value is an address.
- `str` is a pointer constant
  - the value of `str` cannot be changed by assignment.
- `str`` points`` to the memory location of a `char`.

```
'\0'      'o'      'l'      'l'      'e'      'H'
```

6000
Different Views

- two representations – same thing

\[
\begin{array}{cccccc}
\text{H} & \text{e} & \text{l} & \text{l} & \text{o} & \backslash 0 \\
\end{array}
\]
Using a Pointer to Access the Elements of a String

```c
char msg[ ] = "Hello";
char* ptr;
ptr = msg;  // recall that msg == &msg[ 0 ]
*ptr = 'M' ;
ptr++;     // increments the address in ptr
*ptr = 'a';
```
// Requires: str is a null-terminated string
// Effects: returns length of str (not counting '\0')

int stringLength ( char str[] )
{

}

// Requires: str is a null-terminated string
// Effects: returns length of str (not counting '\0')

int stringLength ( char str[] )
{

    for ( ; *p != 0; p++ )
    {
    
    }

}
// Requires: str is a null-terminated string
// Effects: returns length of str (not counting ‘\0’)

int stringLength ( char str[] )
{

    for ( char* p = str; *p != 0; p++)
    {

    }

}
// Requires: str is a null-terminated string
// Effects: returns length of str (not counting '\0')

int stringLength ( char str[] ) {
    int count = 0;
    for ( char* p = str; *p != 0; p++ )
    {
        count ++;
    }
    return count;
}
// Requires: str is a null-terminated string
// Effects: returns length of str (not counting '\0')

int stringLength ( char str[] )
{

    for ( char* p = str; *p != 0; p++ )
    {

    }

}
// Requires: str is a null-terminated string
// Effects: returns length of str (not counting '\0')

int stringLength ( char str[] )
{
    int count = 0;

    for ( char* p = str; *p != 0; p++ )
    {
        count ++;
    }

    return count;
}
int stringLength (char str[]) {
    int count = 0;
    for (char* p = str; *p != 0; p++) {
        count ++;
    }
    return count;
}
### Some C++ Pointer Operations

**Precedence**

<table>
<thead>
<tr>
<th>Higher</th>
<th>-&gt; .</th>
<th>(selecting class elements)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary:</td>
<td>++  <code>--</code>  <code>!</code>  <code>*</code>  <code>&amp;</code>  <code>new</code>  <code>delete</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Increment, Decrement, NOT, Dereference, Address-of, Allocate, Deallocate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*  <code>/</code>  <code>%</code></td>
<td>Mult/Div/Modulus</td>
</tr>
<tr>
<td></td>
<td><code>+</code>  <code>-</code></td>
<td>Add Subtract</td>
</tr>
<tr>
<td></td>
<td><code>&lt;</code>  <code>&lt;=</code>  <code>&gt;</code>  <code>&gt;=</code></td>
<td>Relational operators</td>
</tr>
<tr>
<td></td>
<td><code>==</code>  <code>!=</code></td>
<td>Tests for equality, inequality</td>
</tr>
<tr>
<td></td>
<td><code>=</code></td>
<td>Assignment</td>
</tr>
<tr>
<td>Lower</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The NULL Pointer

• There is a pointer constant 0 called the “null pointer”
  – denoted by NULL in header file cstddef.

• But NULL is not memory address 0.

• NOTE: It is an error to dereference a pointer whose value is NULL.
  – Such an error may cause your program to crash,
  – or behave erratically.
Check for NULL

- It is the programmer’s job to check for this.

```c
while (ptr != NULL)
{
    ...  // ok to use *ptr here
}
```

```c
while (ptr != 0)
{
    ...  // ok to use *ptr here
}
```