CS 52 : C++ Programming

Howard A. Stahl
Agenda

- Pointers
- Dynamic Arrays
Pointers

• Pointers Are A Very Important But Hard To Understand Area Of C++
• Exactly Identical To C Pointers
• Pointers Enable Very Sophisticated Operations
  – dynamic data structures that grow in size over time
  – much more flexible operations and representations
Revisiting Lvalues And Rvalues

- C++ Supports Two Kinds Of Expressions
- Lvalues
  - expressions which can be evaluated and modified
- Rvalues
  - expressions which can only be evaluated
Lvalue And Rvalue Examples

• Lvalue Examples:
  – A Variable Name
    int a;
  – An Array Index
    array[0]

• Rvalue Examples:
  – Literal Constants
    5.14e4
  – Arithmetic Expressions
    5 * a
Lvalues

• An Lvalue Actually Refers To A Location In Memory
  – we conveniently refer it by name
  `int a = 12;`
Lvalues

- An Lvalue Actually Refers To A Location In Memory
  - we conveniently refer it by name

```c
int a = 12;
```

<table>
<thead>
<tr>
<th>MEMORY ADDRESS</th>
<th>1000</th>
<th>1004</th>
<th>...</th>
</tr>
</thead>
</table>

Lvalues

• An Lvalue Actually Refers To A Location In Memory
  – we conveniently refer it by name
  int a = 12;

\[
\begin{array}{c|c}
\text{MEMORY ADDRESS} & \text{a} \\
1000 & \\
1004 & \ldots
\end{array}
\]
Lvalues

• An Lvalue Actually Refers To A Location In Memory
  – we conveniently refer it by name
  int a = 12;

MEMORY
ADDRESS
1000
1004
12
a

...
Pointer Variables

• A Pointer Variable Contains The Address Of A Variable
Pointer Variables

• A Pointer Variable Contains The Address Of A Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
```
Pointer Variables

- A Pointer Variable Contains The Address Of A Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
```

<table>
<thead>
<tr>
<th>MEMORY ADDRESS</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1004</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Pointer Variables

• A Pointer Variable Contains The Address Of A Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
```

MEMORY ADDRESS

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>a</td>
</tr>
<tr>
<td>1004</td>
<td>...</td>
</tr>
</tbody>
</table>
Pointer Variables

• A Pointer Variable Contains The Address Of A Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
```

MEMORY ADDRESS

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>12</td>
</tr>
<tr>
<td>1004</td>
<td>a</td>
</tr>
</tbody>
</table>

...
Pointer Variables

- A Pointer Variable Contains The Address Of A Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
```

![Diagram showing the relationship between `intPtr` and the memory address of `a`]

- `intPtr` contains the address of `a`.
- The memory address of `a` is stored in `intPtr`.
- `intPtr` points to the location in memory where the value `12` is stored.
- The diagram illustrates the memory allocation for `a` and the address stored in `intPtr`.
Pointer Variables

• A Pointer Variable Contains The Address Of A Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
```

![Diagram showing a pointer variable containing the address of a variable](image)
A Real-Life Example

• Consider My Car
A Real-Life Example

• Consider My Car

• We Can Identify It In Many Ways
A Real-Life Example

• Consider My Car

• We Can Identify It In Many Ways
  – VIN # 123456789
  – The third car over from that motorcycle
  – The one next to yours
A Real-Life Example

• Consider My Car

• We Can Identify It In Many Ways
  – VIN # 123456789
  – The third car over from that motorcycle
  – The one next to yours

These Are Pointers!
Pointer Variables

• Like Any Kind Of Variable, Pointers Must Be Declared: *typename varName;
Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: `typename* varName;`
  ```
  double d = 13.1;
  double* dPtr;
  dPtr = &d;
  ```
Pointer Variables

• Like Any Kind Of Variable, Pointers Must Be Declared: `typename* varName;`
  ```
  double d = 13.1;
  double* dPtr;
  dPtr = &d;
  ```
Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: `typename* varName;`
  
  ```
  double d = 13.1;
  
  double* dPtr;
  
  dPtr = &d;
  ```

MEMORY ADDRESS

<table>
<thead>
<tr>
<th>2000</th>
<th>2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>
Pointer Variables

• Like Any Kind Of Variable, Pointers Must Be Declared: `typename* varName;`

```
double d = 13.1;
double* dPtr;
dPtr = &d;
```

<table>
<thead>
<tr>
<th>MEMORY ADDRESS</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>13.1</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Pointer Variables

• Like Any Kind Of Variable, Pointers Must Be Declared: `typename* varName;`

  ```
  double d = 13.1;
  double* dPtr;
  dPtr = &d;
  ```

```
MEMORY ADDRESS
2000
2008

```
Pointer Variables

• Like Any Kind Of Variable, Pointers Must Be Declared: `typename* varName;`

```c
double d = 13.1;
double* dPtr;
dPtr = &d;
```

Diagram:
- `dPtr` points to `d`.
- `d` is a memory address containing the value 13.1.
- The diagram shows the memory address for `d` as 2000 and the value stored as 13.1.
Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: `typename* varName;`

```c
double d = 13.1;
double* dPtr;
dPtr = &d;
```

The Pointer Is An Address We Can Change
Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: `typename* varName;`
  
  ```
  double d = 13.1;
  double* dPtr;
  dPtr = &d;
  ```

The Pointer Is An Address We Can Change

We Can Change What The Pointer Points To
Pointer Variables

- Once declared, a pointer points to only a certain kind of type.
- The thing the pointer points to is called its’ referent.
- The thing the pointer points to is like any other variable of the pointer’s type.
Pointer Dereferencing

• The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable
Pointer Dereferencing

• The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```
Pointer Dereferencing

• The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```

Memory address diagram:
- `intPtr` points to the memory location of `a` at address 1000.
- Value of `*intPtr` is 5, stored at address 1004.
Pointer Dereferencing

• The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```
Pointer Dereferencing

• The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

\[
\begin{align*}
\text{int } a &= 12; \\
\text{int* } \text{intPtr;} \\
\text{intPtr} &= &\& a; \\
*\text{intPtr} &= 5;
\end{align*}
\]
Pointer Dereferencing

• The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```
Pointer Dereferencing

• The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```
• The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```c
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```
Pointer Dereferencing

• * Before A Pointer Variable Is The Dereference Or Indirection Operator
  – it traverses the pointer to access what is being pointed to
Understanding Pointers

• Pointers Are Tricky!
  – keep track of the pointer
  – what is being pointed to
Pointer Assignment

- = Operator Works With Pointers
Pointer Assignment

• = Operator Works With Pointers

```c
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
*p2 = 5;
```
Pointer Assignment

• = Operator Works With Pointers

```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
*p2 = 5;
```
Pointer Assignment

• = Operator Works With Pointers

```c
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```

```
MEMORY
ADDRESS
1000
1004
```

\[ a \] 1000
\[ b \] 1004

\[ a \] 1000
\[ b \] 1004

\[ a \] 1000
\[ b \] 1004
Pointer Assignment

• = Operator Works With Pointers

```c
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```
• = Operator Works With Pointers

\[
\begin{align*}
\text{int } a &= 12, b = 20; \\
\text{int* } p1, *p2; \\
p1 &= &a; \\
p2 &= &b; \\
p2 &= p1; \\
*p2 &= 5;
\end{align*}
\]
Pointer Assignment

• = Operator Works With Pointers

```c
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
*p2 = 5;
```
Pointer Assignment

- = Operator Works With Pointers

```c
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
*p2 = 5;
```
Pointer Assignment

• = Operator Works With Pointers

```c
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
*p2 = 5;
```
Pointer Assignment

• = Operator Works With Pointers

```c
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
*p2 = 5;
```
Pointer Assignment

• = Operator Works With Pointers
• = Operator Changes What The Pointer Variable Points To
Time For Our Next Demo!

• PointerEquals.cpp

(See Handout For Example 1)
Summarizing Our Next Demo!

- = Operator Changes The Address Of What Is Being Pointed To
new Operators

• Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The new Operator

```c
int* p1;
p1 = new int;
*p1 = 10;
```
new Operators

• Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The `new` Operator

```cpp
int* p1;
p1 = new int;
*p1 = 10;
```

MEMORY ADDRESS

```
1000
1004
```
new Operators

• Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The new Operator

```c
int* p1;
p1 = new int;
*p1 = 10;
```
new Operators

• Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The new Operator

```c
int* p1;
p1 = new int;
*p1 = 10;
```
new Operators

• Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The new Operator

```c
int* p1;
p1 = new int;
*p1 = 10;
```
new Operators

- Pointers Can Work With Any Class Type
- `new` Operator Makes A Constructor Call;

```cpp
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
```
new Operators

• Pointers Can Work With Any Class Type
• new Operator Makes A Constructor Call;

```c
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
```

```c
bPtr
```
new Operators

- Pointers Can Work With Any Class Type
- `new` Operator Makes A Constructor Call;

```cpp
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
```
delete Operators

• All Dynamic Variables Must Be delete’d To Recycle Memory Used

```cpp
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```
delete Operators

• All Dynamic Variables Must Be delete’d To Recycle Memory Used

bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
cout << (*bPtr).balance();
... 
delete bPtr;  // bPtr
delete Operators

• All Dynamic Variables Must Be delete’d To Recycle Memory Used

```cpp
bankAccount* bPtr;
bPtr = new bankAccount(“howie”, 10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```
delete Operators

• All Dynamic Variables Must Be delete’d To Recycle Memory Used

```cpp
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```
delete Operators

• All Dynamic Variables Must Be delete’d To Recycle Memory Used

```cpp
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
cout << (*bPtr).balance();
...
delte bPtr;  
```
Pointer Basics

- A Pointer Must Point To Something Before You Dereference The Pointer
- Once Deleted, You Cannot Dereference The Pointer Anymore
- The -> Operator Is A Shorthand For (*ptr_variable).member
Dynamic Arrays

- `new` and `delete` Operators Support Dynamic Arrays

typedef double* doublePtr;
doublePtr d;
int n;
n = ...;
d = new double[ n ];
fill_up( d[0] );
delete [] d;
Dynamic Arrays

- new And delete Operators Support Dynamic Arrays

typedef double* doublePtr;
doublePtr d;
int n;
n = ...;
d = new double[ n ];
fill_up( d[0] );
delete [] d;

Array Size Is Not A Fixed Constant
Dynamic Arrays

- `new` and `delete` Operators Support Dynamic Arrays

```cpp
typedef double* doublePtr;
doublePtr d;
int n;
n = ...;
d = new double[ n ];
fill_up( d[0] );
delete [] d;
```

Array Size Is Not A Fixed Constant

Dynamic Array Is Used Like Any Other Array
Dynamic Arrays

- new and delete Operators Support Dynamic Arrays

```c
typedef double* doublePtr;
doublePtr d;
int n;
n = ...
d = new double[ n ];
fill_up( d[0] );
delete [] d;
```

Array Size Is Not A Fixed Constant
Dynamic Array Is Used Like Any Other Array
Note delete Syntax
Observation

• Dynamic Arrays Are A Useful Way To Process DataSets Of Unknown Size
• Dynamic Arrays Of Class Type Is A Common Construct
Summary

• Pointers
• Dynamic Arrays