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
  int a = 12;

  MEMORY ADDRESS
  1000
  1004

Lvalues

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

  MEMORY ADDRESS
  1000
  1004

Lvalues

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

  MEMORY ADDRESS
  1000
  1004

Lvalues

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

  MEMORY ADDRESS
  1000
  1004
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;
MEMORY ADDRESS
1000
1004
```

```c
12
```

```
1000
1004
```
Pointer Variables

• A Pointer Variable Contains The Address Of A Variable

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

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

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;`

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

MEMORY ADDRESS
```
2000 13.1 d
2008 ...
```
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

```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**

- 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

```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;
```

Memory address `1000` for `a` and `1004` for `b`.
Pointer Assignment

• = Operator Works With Pointers

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

**Example Diagram:**

Diagram showing memory addresses and values accessed by pointers `p1` and `p2`.
Pointer Assignment

• = Operator Works With Pointers
• = Operator Changes What The Pointer Variable Points To

```c
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```
Time For Our Next Demo!

- PointerEquals.cpp

Summarizing Our Next Demo!

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

Textbook Example

---

Display m.a Uses of the Assignment Operator with Pointer Variables

p1 = p2;  
Before:  
p1  
p2  
After:  
```
```

```
```

p1 = &p2;  
Before:  
p1  
p2  
After:  
```
```

```
```

---
new Operators

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

```c
int* pl;
pl = new int;
*pl = 10;
```

new Operators

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

```c
int* pl;
pl = new int;
*pl = 10;
```

new Operators

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

```c
int* pl;
pl = new int;
*pl = 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;
```

Textbook Example

```c
int main()
{
    int p1, *p1;
    p1 = new int;
    *p1 = 10;
    p1 = 5;
    *p1 = 15;
}
```
Textbook Example

```c
int main() {
    int p = 10;
    int q = 20;
    int r = 30;
    printf("p = \%d, q = \%d, r = \%d\n", p, q, r);
}
```

Another Textbook Example

```c
#include <stdio.h>

void display(int *p) {
    int temp = *p;
    printf("Before call to function \"p = \%d\n\", temp);
    *p = 30;
    printf("After call to function \"p = \%d\n\", temp);
}

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 10;
    display(p);
    free(p);
    return 0;
}
```
Another Textbook Example

```cpp
16  sneaky(p):
17     cout << "after call to function \n\n = "
18     cout << *p << endl;
19     return 0;
20  int
21     void sneaky(int* temp)
22     {  
23         *temp = 99;
24         cout << "inside function call \n\n = "
25         << *temp << endl;
26     }
```

Another Textbook Example

- Display no. 5. The function call sneaky(p):

1. Before call to sneaky:
   - `p` is at 77
2. Value of `p` is plugged in for `temp`:
   - `p` points to `temp`
3. Change made to `*temp`:
   - `*temp` is 99
4. After call to sneaky:
   - `p` points to `temp` again

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;

```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

```cpp
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;  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;  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; 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

```cpp
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

```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

Understanding Arrays Now...

• Arrays Are Pointer Variables
  – The Array Variable Points At The First Indexed Variable
• Example:
  ```
  int array[ 10 ];
  int * p;
  ```
• Arrays Are Pointer Variables
  – The Array Variable Points At The First Indexed Variable

Example:
```c
int array[10];
int * p;
p = array;   // LEGAL!
```

• Arrays Are Pointer Variables
  – The Array Variable Points At The First Indexed Variable

Example:
```c
int array[10];
int * p;
p = array[5]; // LEGAL!
```

• Arrays Are Pointer Variables
  – The Array Variable Points At The First Indexed Variable

Example:
```c
int array[10];
int * p;
array = p;   // ILLEGAL!
```
Understanding Arrays Now…

• Arrays Are Pointer Variables
  – The Array Variable Points At The First Indexed Variable

• Example:
  ```c
  int array[10];
  int * p;
  array = p;   // ILLEGAL!
  ```
  
  `Declared Array Is A CONSTANT Pointer!`

• Arrays Are Pointer Variables
  – The Array Variable Points At The First Indexed Variable

  ```c
  const int * The Type Is const int *
  ```
  
  `Declared Array Is A CONSTANT Pointer!`

• Arrays Are Pointer Variables
  – The Array Variable Points At The First Indexed Variable

  ```c
  A Declared Array Variable Is A Constant Pointer Allocated In Memory Already And Variable Must Point There And Cannot Be Changed
  ```

  ```c
  – The Address Cannot Be Changed But The Referent Can Be Changed
  ```
Returning Arrays From A Function

• You Cannot Return An Array From A Function

int [ ] someFunction( );

// NOT LEGAL!
Returning Arrays From A Function

• You Cannot Return An Array From A Function
  int [ ] someFunction( );
  // NOT LEGAL!

• But You Can Return An int * From A Function
  int * someFunction( );
  // LEGAL
  // new array in the function

Returning Arrays From A Function

• You Cannot Return An Array From A Function
  int [ ] someFunction( );
  // NOT LEGAL!

• But You Can Return An int * From A Function
  int * someFunction( );
Pointer Arithmetic

- You Can Perform Arithmetic On Pointer Addresses
- You Can Use +, -, ++ Or -- But Not * Or /

```cpp
double * arr = new double[5];
arr + 1 Evaluates To arr[1]
arr + 2 Evaluates To arr[2]
```

Using Pointer Arithmetic, The Following Code Is Equivalent:
```cpp
double * arr = new double[5];
for (int i=0; i<5; i++)
    { cout << arr[i] << " \n"; }
```
**Pointer Arithmetic**

- Using Pointer Arithmetic, The Following Code Is Equivalent:
  
  ```cpp
  double * arr = new double[5];
  for (int i=0; i<5; i++)
  { cout << arr[i] << " "; }
  double * arr = new double[5];
  for (int i=0; i<5; i++)
  { cout << *(arr + i) << " "; }
  ```

**Summary**

- Pointers
- Dynamic Arrays
- Pointer Arithmetic