UCLA

CS 31:
Introduction To Computer Science I

Howard A. Stahl

UCLA

Agenda

- Pointers
- Dynamic Arrays
- Pointer Arithmetic

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

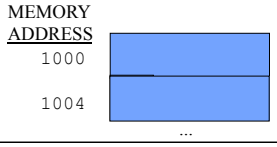
- An Lvalue Actually Refers To A Location In Memory
  - we conveniently refer it by name

```
int a = 12;
```

Lvalues

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

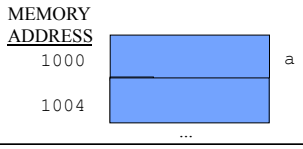
```
int a = 12;
```



Lvalues

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

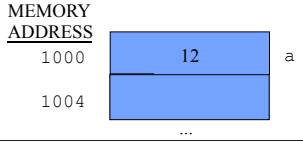
```
int a = 12;
```



Lvalues

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

```
int a = 12;
```



Pointer Variables

- A Pointer Variable Contains The Address Of A Variable

Pointer Variables

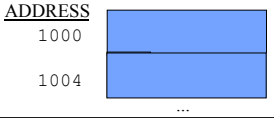
- A Pointer Variable Contains The Address Of A Variable

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

Pointer Variables

- A Pointer Variable Contains The Address Of A Variable

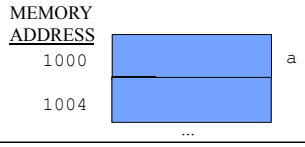
```
int a = 12;
int* intPtr;
intPtr = &a;
```



Pointer Variables

- A Pointer Variable Contains The Address Of A Variable

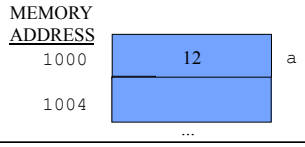
```
int a = 12;
int* intPtr;
intPtr = &a;
```



Pointer Variables

- A Pointer Variable Contains The Address Of A Variable

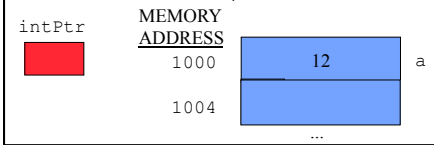
```
int a = 12;
int* intPtr;
intPtr = &a;
```



Pointer Variables

- A Pointer Variable Contains The Address Of A Variable

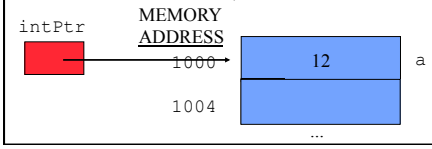
```
int a = 12;
int* intPtr;
intPtr = &a;
```



Pointer Variables

- A Pointer Variable Contains The Address Of A Variable

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



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

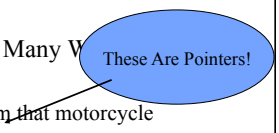
- Consider My Car

- 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 W
  - VIN # 123456789
  - The third car over from that motorcycle
  - The one next to yours



Pointer Variables

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

Pointer Variables

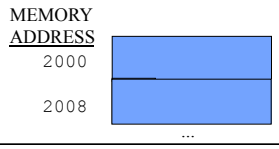
- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

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

Pointer Variables

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

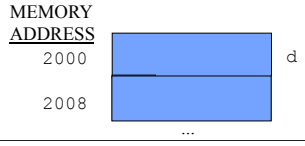
```
double d = 13.1;
double* dPtr;
dPtr = &d;
```



Pointer Variables

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

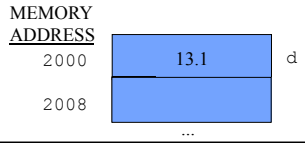
```
double d = 13.1;
double* dPtr;
dPtr = &d;
```



Pointer Variables

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

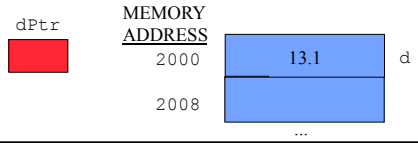
```
double d = 13.1;
double* dPtr;
dPtr = &d;
```



Pointer Variables

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

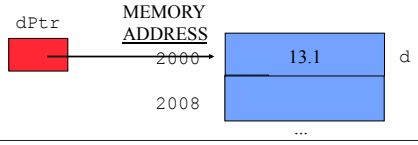
```
double d = 13.1;
double* dPtr;
dPtr = &d;
```



Pointer Variables

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

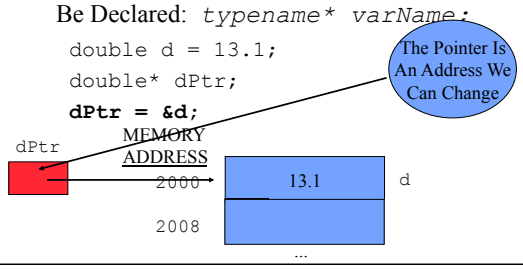
```
double d = 13.1;
double* dPtr;
dPtr = &d;
```



Pointer Variables

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

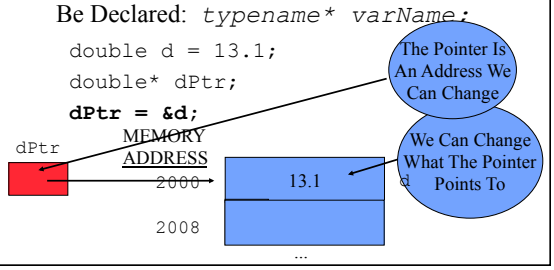
```
double d = 13.1;
double* dPtr;
dPtr = &d;
```



Pointer Variables

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

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



Pointer Variables

- 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

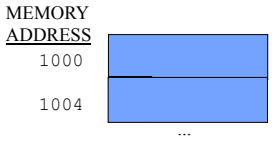
- The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

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

Pointer Dereferencing

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

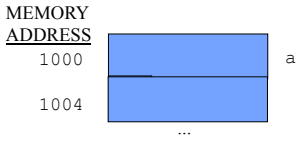
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Pointer Dereferencing

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

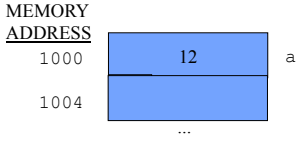
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Pointer Dereferencing

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

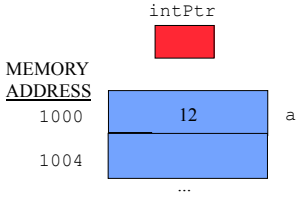
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Pointer Dereferencing

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

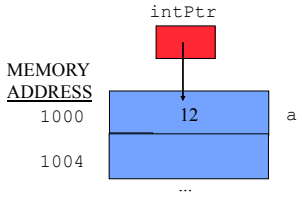
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Pointer Dereferencing

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

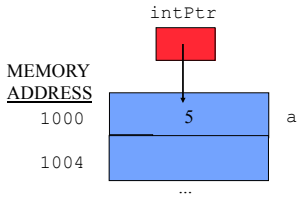
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Pointer Dereferencing

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

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



Pointer Dereferencing

- * Before A Pointer Variable Is The Derefence 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

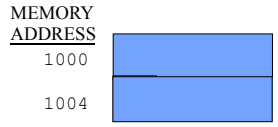
- = Operator Works With Pointers

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

Pointer Assignment

- = Operator Works With Pointers

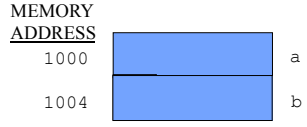
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

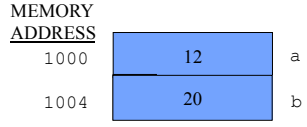
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

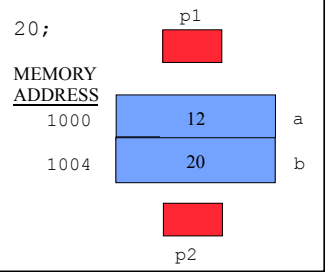
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

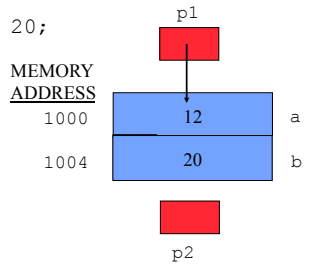
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

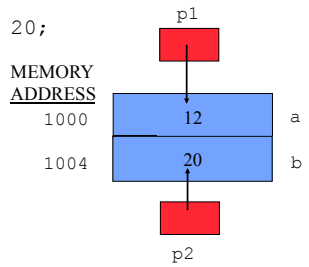
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

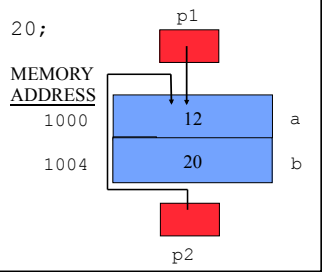
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

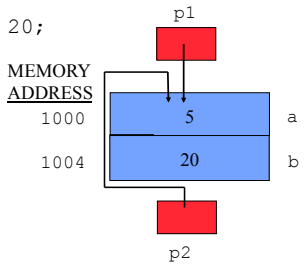
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

```
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

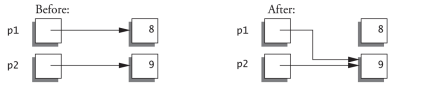
Summarizing Our Next Demo!

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

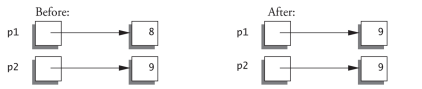
Textbook Example

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```



new Operators

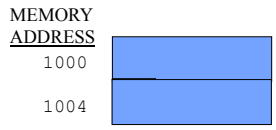
- Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The new Operator

```
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

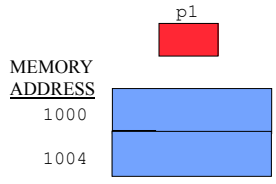
```
int* p1;
p1 = new int;
*p1 = 10;
```



new Operators

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

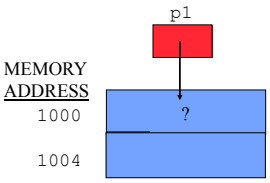
```
int* p1;
p1 = new int;
*p1 = 10;
```



new Operators

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

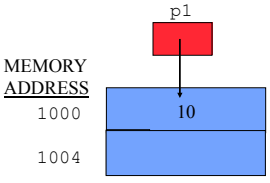
```
int* p1;
p1 = new int;
*p1 = 10;
```



new Operators

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

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



Textbook Example

Display 10.2 **Basic Pointer Manipulations**

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;

5  int main( )
6  {
7      int *p1, *p2;

8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;

13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;
```

Textbook Example

```
16      p1 = new int;
17      *p1 = 88;
18      cout << "*p1 == " << *p1 << endl;
19      cout << "*p2 == " << *p2 << endl;

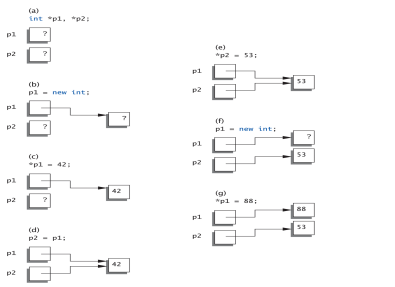

20      cout << "Hope you got the point of this example!\n";
21      return 0;
22  }
```

SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Textbook Example

Display 10.3 **Explanation of Display 10.2**



Another Textbook Example

Display 10.4 **A Call-by-Value Pointer Parameter**

```
1   //Program to demonstrate the way call-by-value parameters
2   //behave with pointer arguments.
3   #include <iostream>
4   using std::cout;
5   using std::cin;
6   using std::endl;

7   typedef int* IntPointer;

8   void sneaky(IntPointer temp);

9   int main( )
10  {
11      IntPointer p;

12      p = new int;
13      *p = 77;
14      cout << "Before call to function *p == "
15           << *p << endl;
```

Another Textbook Example

```
16      sneaky(p);

17      cout << "After call to function *p == "
18           << *p << endl;

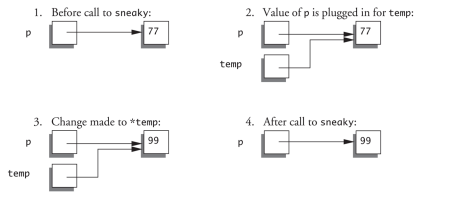
19      return 0;
20  }
21  void sneaky(IntPointer temp)
22  {
23      *temp = 99;
24      cout << "Inside function call *temp == "
25           << *temp << endl;
26  }
```

SAMPLE DIALOGUE

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

Another Textbook Example

Display 10.5 **The Function Call sneaky(p);**



`new` Operators

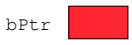
- Pointers Can Work With Any Class Type
- `new` Operator Makes A Constructor Call;

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

new Operators

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

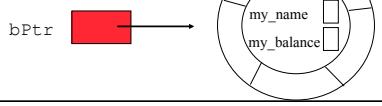
```
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
```



new Operators

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

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



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;
```

delete Operators

- All Dynamic Variables Must Be delete'd To Recycle Memory Used

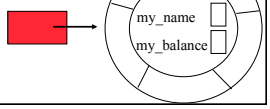
```
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```

bPtr 

delete Operators

- All Dynamic Variables Must Be delete'd To Recycle Memory Used

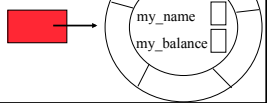
```
bankAccount* bPtr;
bPtr = new bankAccount("howie", 10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```

bPtr 

delete Operators

- All Dynamic Variables Must Be delete'd To Recycle Memory Used

```
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

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

```
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

```
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…

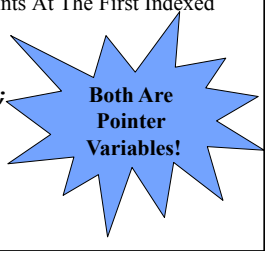
- Arrays Are Pointer Variables
  - The Array Variable Points At The First Indexed Variable
- Example:

```
int array[ 10 ];
int * p;
```

Understanding Arrays Now…

- Arrays Are Pointer Variables
  - The Array Variable Points At The First Indexed Variable
- Example:

```
int array[ 10 ];
int * p;
```



Understanding Arrays Now…

- Arrays Are Pointer Variables
  - The Array Variable Points At The First Indexed Variable
- Example:

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

Understanding Arrays Now…

- Arrays Are Pointer Variables
  - The Array Variable Points At The First Indexed Variable
- Example:

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

Understanding Arrays Now…

- Arrays Are Pointer Variables
  - The Array Variable Points At The First Indexed Variable
- Example:

```
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:

```
int array[ 10 ];
int * p;
array = p;
```

Declared Array Is A CONSTANT Pointer!

Understanding Arrays Now…

- Arrays Are Pointer Variables
  - The Array Variable Points At The First Indexed Variable
- Exam

The Type Is `const int *`

Declared Array Is A CONSTANT Pointer!

Understanding Arrays Now…

- Arrays Are Pointer Variables
  - The Array Variable Points At The First Indexed Variable
  - A Declared Array Variable Is A Constant Pointer Allocated In Memory Already And Variable Must Point There And Cannot Be Changed
  - The Address Cannot Be Changed But The Referent Can Be Changed

Returning Arrays From A Function

- You Cannot Return An Array From A Function

Returning Arrays From A Function

- You Cannot Return An Array From A Function

```
int [ ] someFunction( );
```

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

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( );
```

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
```

Pointer Arithmetic

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

Pointer Arithmetic

- You Can Perform Arithmetic On Pointer Addresses
- You Can Use +, -, ++ Or -- But Not * Or /
- `double * arr=new double[ 5 ];`
- `arr` Evaluates To `arr[ 0 ]`
  `arr + 1` Evaluates To `arr[ 1 ]`
  `arr + 2` Evaluates To `arr[ 2 ]`

Pointer Arithmetic

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

Pointer Arithmetic

- Using Pointer Arithmetic, The Following Code Is Equivalent:
- ```
  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