

Pointers, Vectors, and Lists

Michael C. Hackett
Computer Science Department

Community
College
of Philadelphia

Lecture Topics

- Pointers
 - Declaration and Initialization
 - Null Pointers
 - Pointer Operations
 - Dereferencing Pointers
- Functions and Pointers
 - Passing by value
 - Passing by reference (reference args)
 - Passing by reference (pointer args)
- Pointer Arithmetic
- Pointers and Arrays
- Vectors
 - Declaration and Initialization
 - Adding data to a Vector
 - Removing data from a Vector
 - Inserting data into a Vector
- Lists
 - Declaration and Initialization
 - Adding data to a List
 - Removing data from a List
 - Inserting data into a List
- Complexity Comparison

Pointer Variables

- Pointer variables contain memory addresses as their values.
 - A normal variable directly references the value at a memory address.
- Pointers *indirectly* reference a value.
 - Referencing a value using a pointer is called **indirection**.

Declaring Pointers

- A pointer is declared much like a variable.
 - The pointer variable name is preceded by a *
 - *Indirection operator/Dereferencing operator*

```
int *examplePtr;
```

- It's good practice to add a "Ptr" suffix so pointers are easily identified in the source code, but this isn't a requirement.

Initializing (Null) Pointers

- Pointers can be initialized with 0, NULL (a constant from iostream) or an address.
 - The NULL constant is assigned 0; 0 is conventionally used in C++.

```
int *examplePtr;  
*examplePtr = 0;
```

```
double *example2Ptr = 0;
```

```
int *example3Ptr = NULL;
```

- When 0 is assigned, it is converted to a pointer of the appropriate type.

Initializing Pointers

- The address operator **&** returns the memory address of a variable.

```
int x = 7;  
int *xPtr;
```

```
xPtr = &x;
```

- In the third line, the address operator returns the address of the x variable and assigns it to the xPtr pointer.
 - **Only addresses can be assigned to pointer variables.**

Pointer Operators

```
int *xPtr = 0;
```

```
cout << xPtr << endl;  
cout << *xPtr << endl;
```

Output:

0

<Program crashes>

- Nothing to dereference from xPtr (it is null)
 - The “0” printed is the address, not a value.

Pointer Operators

```
int x = 7;
```

```
cout << x << endl;
```

```
cout << &x << endl;
```

Output:

7

0x6ffe4c

Pointer Operators

```
int x = 7;  
int *xPtr;
```

```
xPtr = &x;
```

```
cout << "x = " << x << endl;  
cout << "x addr = " << &x << endl;  
cout << "xPtr = " << xPtr << endl;  
cout << "xPtr deref = " << *xPtr << endl;
```

Output:

```
x = 7  
x addr = 0x6ffe4c  
xPtr = 0x6ffe4c  
xPtr deref = 7
```

Dereferencing Pointers

- The last line on the previous slide...

```
cout << "xPtr deref = " << *xPtr << endl;
```

demonstrates **dereferencing** a pointer.

- Attempting to dereference a non-pointer variable is a syntax error.

Dereferencing Pointers

- Forgetting to dereference a pointer when it is necessary to do so will cause an error.

```
int x = 3;  
int *xPtr;  
xPtr = &x;  
int y = 5;  
int z = y + xPtr;
```

- The last line should read:

```
int z = y + *xPtr;
```

Functions and Pointers

- Three ways to call a function in C++:
 - Pass-by-value
 - Example prototype: **int cubicArea(int)**
 - Pass-by-reference with reference arguments
 - Example prototype: **void cubicArea(int &)**
 - Pass-by-reference with pointer arguments
 - Example prototype: **void cubicArea(int *)**

Pass-by-Value

- When data is passed by value to a function, the function's parameter get a copy of the value passed to it.

```
int main() {  
    int x = 5;  
    test(x);  
    cout << "x = " << x << endl;  
}
```

Output:

x = 5

```
void test(int v) {  
    v++;  
}
```



Increments v, which is x's *value*.
This does not alter x back in the main function.

Pass-by-Reference

- When data is passed by reference to a function, the function's parameter gets the reference/address of the value passed to it.

```
int main() {  
    int x = 5;  
    test(x);  
    cout << "x = " << x << endl;  
}
```

Output:

x = 6

```
void test(int &v) {  
    v++;  
}
```

Increments v, which is x's *reference/address*.



Pass-by-Value

```
int main() {  
    int x = 5;  
    int area = cubicArea(x);  
  
    cout << "The area is " << area << endl;  
    cout << "x = " << x << endl;  
}
```

```
int cubicArea(int value) {  
    int result = value * value * value;  
    return result;  
}
```

Output:

```
The area is 125  
x = 5
```

Pass-by-Reference (Reference Arguments)

```
int main() {  
    int x = 5;  
    cubicArea(x);  
  
    cout << "x = " << x << endl;  
}  
  
void cubicArea(int &value) {  
    value = value * value * value;  
}
```

Output:

x = 125

Pass-by-Reference (Pointer Arguments)

```
int main() {  
    int x = 5;  
    cubicArea(&x);  
  
    cout << "x = " << x << endl;  
}
```

```
void cubicArea(int *valuePtr) {  
    *valuePtr = *valuePtr * *valuePtr * *valuePtr;  
}
```

Output:

x = 125

A swap function without pointers

```
int main() {  
    int array[] = {1, 2, 4, 3, 5};  
    swap(array, 2, 3);  
}  
  
void swap(int[] a, int i1, int i2) {  
    int temp = a[i1];  
    a[i1] = a[i2];  
    a[i2] = temp;  
}
```

A swap function with pointers

```
int main() {  
    int array[] = {1, 2, 4, 3, 5};  
    swap(&array[2], &array[3]);  
}
```

```
void swap(int *n1, int *n2) {  
    int temp = *n1;  
    *n1 = *n2;  
    *n2 = temp;  
}
```

- This would swap any two ints, not just ints in an array.

Pointer Arithmetic

- Pointers can be used with arithmetic operations like addition and subtraction.
- When a pointer is added to or subtracted from, it is not incremented or decremented by the integer value, but by the number of bytes to which the object points.

Pointer Arithmetic

```
int x = 7;  
int *xPtr = &x;  
  
cout << "xPtr = " << xPtr << endl;  
  
xPtr += 1;  
  
cout << "xPtr = " << xPtr << endl;
```

Output:

```
xPtr = 0x6ffe44  
xPtr = 0x6ffe48
```

Pointers and Arrays

- Pointers can be used to do any operations involving subscripts.
 - **An array variable is actually a pointer.**
 - It references the address of the first element.

```
int x[] = {2, 4, 6, 8, 10};  
int *xPtr = x;
```

```
cout << "xPtr = " << *xPtr << endl;
```

```
xPtr += 1;
```

```
cout << "xPtr = " << *xPtr << endl;
```

Output:

xPtr = 2

xPtr = 4

Pointers and Arrays

```
int x[] = {2, 4, 6, 8, 10};  
int *xPtr = x;  
  
cout << "xPtr = " << *xPtr << endl;  
  
cout << "xPtr + 2 = " << *(xPtr + 2) << endl;  
cout << "xPtr = " << *xPtr << endl;
```

Output:

xPtr = 2

xPtr + 2 = 6

xPtr = 2

Vectors

- Vectors are container objects (like arrays) that dynamically grow or shrink in size (unlike arrays).
- Uses contiguous memory, like arrays.
- Including the vector header is required:

`#include<vector>`

Declaring a Vector

- Vectors are declared using the following syntax:
 - This declares a vector of ints.

```
vector<int> v;
```

Adding to a Vector

- Values are added to the end of the sequence.
 - The first value in the vector is the “front”
 - The last value in the vector is the “back”
- The vector’s `push_back()` function is used to add a value to the end of the series.

```
v.push_back(4);  
v.push_back(2);  
v.push_back(8);
```

Getting the Length of a Vector

- The vector's `size()` function is used to retrieve the number of elements in the sequence.

```
vector<int> v;  
  
v.push_back(4);  
v.push_back(2);  
v.push_back(8);  
  
int vLength = v.size();  
cout << "vLength = " << vLength << endl;
```

Output:

```
vLength = 3
```

Retrieving Data from a Vector

- Subscript notation can be used to retrieve or replace existing values in a vector.

```
vector<int> v;
```

```
v.push_back(4);
```

```
v.push_back(2);
```

```
v.push_back(8);
```

```
for(int i = 0; i < v.size(); i++){  
    cout << v[i] << endl;  
}
```

Output:

4

2

8

Retrieving the First Element from a Vector

- The `front()` function retrieves the first element in the sequence.

```
vector<int> v;
```

```
v.push_back(4);
```

```
v.push_back(2);
```

```
v.push_back(8);
```

```
cout << v.front() << endl;
```

Output:

4

Retrieving the Last Element from a Vector

- The `back()` function retrieves the last element in the sequence.

```
vector<int> v;
```

```
v.push_back(4);
```

```
v.push_back(2);
```

```
v.push_back(8);
```

```
cout << v.back() << endl;
```

Output:

8

Removing Data from a Vector

- The `pop_back()` function removes, but does not retrieve the last element in the sequence.
 - “Push” = adding to the sequence
 - “Pop” = removing from the sequence

```
vector<int> v;  
v.push_back(4);  
v.push_back(2);  
v.push_back(8);  
v.pop_back();  
for(int i = 0; i < v.size(); i++){  
    cout << v[i] << endl;  
}
```

Output:

4
2

Removing Data from a Vector

- The `erase()` function removes, but does not retrieve a specific element in the sequence.
 - One parameter: An iterator type: use `v.begin()+n`.

```
vector<int> v;  
v.push_back(4);  
v.push_back(2);  
v.push_back(8);  
v.erase(v.begin()+1);  
for(int i = 0; i < v.size(); i++){  
    cout << v[i] << endl;  
}
```

Output:

4
8

Inserting Data into a Vector

- The `insert()` function inserts a new element into the sequence.
 - Two parameters: An iterator type: use `v.begin()+n`; The value to insert

```
vector<int> v;  
v.push_back(4);  
v.push_back(2);  
v.push_back(8);  
v.insert(v.begin()+1, 7);  
for(int i = 0; i < v.size(); i++){  
    cout << v[i] << endl;  
}
```

Output:

4
7
2
8

Clearing the Vector

- The `clear()` function removes all elements from the sequence.

```
v.clear();
```

Lists

- Lists are container objects (like arrays and vectors) that can dynamically grow or shrink in size (like vectors).
- Lists **do not** use contiguous memory space. (Unlike arrays and vectors)
 - The data can be all over the place, so to speak.
- Including the list header is required :

`#include<list>`

Declaring a List

- Lists are declared using the following syntax:
 - This declares a list of ints.

```
list<int> w;
```

Adding to a List (Back)

- The list's `push_back()` function is used to add a number to the end of the series.

```
w.push_back(4);  
w.push_back(2);  
w.push_back(8);
```

Adding to a List (Front)

- The list's `push_front()` function is used to add a number to the beginning of the series.
 - Can't add to the front of a Vector

```
w.push_front(3);  
w.push_front(5);  
w.push_front(7);
```

Getting the Length of a List

- The list's `size()` function is used to retrieve the number of elements in the sequence.

```
list<int> w;  
w.push_back(4);  
w.push_back(2);  
w.push_back(8);  
w.push_front(3);  
w.push_front(5);  
w.push_front(7);  
int wLength = w.size();  
cout << "wLength = " << wLength << endl;
```

Output:

```
wLength = 6
```

Retrieving Data from a List

- Subscript notation can't be used since the data is not using contiguous memory.
 - Lists (C++'s list anyway) doesn't give us an easy way to access individual elements.
- We will instead need to use an iterator.
 - Essentially, a pointer.
- We can retrieve the element by dereferencing the iterator.

Retrieving Data from a List

```
list<int> w;  
w.push_back(4);  
w.push_back(2);  
w.push_back(8);  
w.push_front(3);  
w.push_front(5);  
w.push_front(7);  
list<int>::iterator t = w.begin();  
for(int i = 0; i < w.size(); i++){  
    cout << *t << endl;  
    t++;  
}
```

Output:

7
5
3
4
2
8

Retrieving the First Element from a List

- The `front()` function retrieves the first element in the sequence.

```
list<int> w;  
w.push_back(4);  
w.push_back(2);  
w.push_back(8);  
w.push_front(3);  
w.push_front(5);  
w.push_front(7);
```

```
cout << w.front() << endl;
```

Output:

7

Retrieving the Last Element from a List

- The `back()` function retrieves the last element in the sequence.

```
list<int> w;  
w.push_back(4);  
w.push_back(2);  
w.push_back(8);  
w.push_front(3);  
w.push_front(5);  
w.push_front(7);  
  
cout << w.back() << endl;
```

Output:

8

Removing Data from a List (Back)

- The `pop_back()` function removes the last element in the sequence.

```
list<int> w;  
w.push_back(4);  
w.push_back(2);  
w.push_back(8);  
w.push_front(3);  
w.push_front(5);  
w.push_front(7);  
  
w.pop_back();  
  
list<int>::iterator t = w.begin();  
for(int i = 0; i < w.size(); i++){  
    cout << *t << endl;  
    t++;  
}
```

Output:

7

5

3

4

2

Removing Data from a List (Front)

- The `pop_front()` function removes the first element in the sequence.

```
list<int> w;  
w.push_back(4);  
w.push_back(2);  
w.push_back(8);  
w.push_front(3);  
w.push_front(5);  
w.push_front(7);  
  
w.pop_front();  
  
list<int>::iterator t = w.begin();  
for(int i = 0; i < w.size(); i++){  
    cout << *t << endl;  
    t++;  
}
```

Output:

5
3
4
2
8

Removing Data from a List

- The `erase()` function removes, but does not retrieve a specific element in the sequence.
 - One parameter: An iterator type: use `v.begin()+n`.

```
list<int> w;  
w.push_back(4);  
w.push_back(2);  
w.push_back(8);  
w.push_front(3);  
w.push_front(5);  
w.push_front(7);
```

```
list<int>::iterator t = w.begin();  
t++;  
w.erase(t);
```

```
t = w.begin();  
for(int i = 0; i < w.size(); i++){  
    cout << *t << endl;  
    t++;  
}
```

Output:

7

3

4

2

8

Inserting Data into a List

- The `insert()` function inserts a new element into the sequence.
 - Two parameters: An iterator type and the value to insert

```
list<int> w;  
w.push_back(4);  
w.push_back(2);  
w.push_back(8);  
w.push_front(3);  
w.push_front(5);  
w.push_front(7);  
  
list<int>::iterator t = w.begin();  
t++;  
w.insert(t, 6);  
  
t = w.begin();  
for(int i = 0; i < w.size(); i++){  
    cout << *t << endl;  
    t++;  
}
```

Output:

7
6
5
3
4
2
8

Clearing the List

- The `clear()` function removes all elements from the sequence.

```
w.clear();
```


Complexity Comparisons

- Retrieval
 - Arrays
 - Example: `int x = a[1];`
 - Complexity: **$O(1)$**
 - Vectors
 - Example: `int y = v[1];`
 - Complexity: **$O(1)$**
 - Lists
 - Example: `list<int>::iterator t = w.begin();`
`t++;`
`int z = *t;`
 - Complexity: **$O(1)$**

Complexity Comparisons

- Appending to back
 - Arrays
 - Not Applicable
 - Vectors
 - Example: `v.push_back(7);`
 - Complexity*: **$O(1)$**
 - *If there is free, contiguous space available. Otherwise, the vector needs a new reallocation of memory. The complexity in that case is **$O(n)$** since the vector's data needs to be copied over.
 - Lists
 - Example: `v.push_back(7);`
 - Complexity: **$O(1)$**

Complexity Comparisons

- Appending to front
 - Arrays
 - Not Applicable
 - Vectors
 - Not Applicable
 - Lists
 - Example: `w.push_front(4);`
 - Complexity: **$O(1)$**

Complexity Comparisons

- Insertion (not front or back)
 - Arrays
 - Not Applicable
 - Vectors
 - Example: `v.insert(v.begin()+3, 7);`
 - Complexity: **$O(n)$**
 - Lists
 - Example: `list<int>::iterator t = w.begin();`
`t += 3;`
`w.insert(7);`
 - Complexity: **$O(1)$**

Complexity Comparisons

- Removal from back
 - Arrays
 - Not Applicable
 - Vectors
 - Example: `int y = v.pop_back();`
 - Complexity: **$O(1)$**
 - Lists
 - Example: `int z = w.pop_back();`
 - Complexity: **$O(1)$**

Complexity Comparisons

- Removal from front
 - Arrays
 - Not Applicable
 - Vectors
 - Example: `v.erase(v.begin());`
 - Complexity*: **$O(n)$**
 - *Needs to move the other elements forward.
 - Lists
 - Example: `int z = w.pop_front();`
 - Complexity: **$O(1)$**

Complexity Comparisons

- Removal from anywhere
 - Arrays
 - Not Applicable
 - Vectors
 - Example: `v.erase(v.begin()+1);`
 - Complexity: **$O(n)$**
 - Lists
 - Example: `list<int>::iterator t = w.begin();`
`t++;`
`w.erase(t);`
 - Complexity: **$O(1)$**

Arrays, Vectors, Lists

- Arrays and Vectors use contiguous space.
 - Lists do not.
- Arrays have fixed lengths.
 - Vectors and Lists do not.
- Lists do not have random access; An iterator is required.
 - Arrays and Vectors do not require the use of an iterator to access values; They can use indexes to retrieve and replace data.
 - While nearly all List operations are constant, any iteration required will be done in linear time.