C++ and Object-Oriented Programming

Pointers, Containers, and Iterators

Pointers, Containers, and Iterators

- Chapter Objectives
  - understand that pointer objects store addresses of other objects
  - use several methods for initializing pointers
    - they can store null or an address
  - use the standard `list` class
    - from the standard template library (STL)
  - use the STL `iterator` class to iterate over a `list` object

Memory Considerations

- In addition to name, state, and operations, every object has a fourth attribute—its address
- With the following initialization, we see that the name (charlie), state (100), and operations (the set of int operations) are known:

```
int charlie = 100; // But where is it stored?
```

Addresses

- An object's address is the actual memory location where the first byte of the object is stored
- The actual memory location is something we have not needed to know about until now.

Static and Dynamic Memory Allocation

- Some objects take a fixed amount of memory at compile time:

  ```
  char    int    double
  ```

- Other objects require varying amounts of memory, which is allocated and deallocated dynamically, that is, at runtime: string (string objects change size during =, >>, and +)
- We usually use pointer objects to allow for these *dynamic* objects

Pointer Objects

- Pointer objects store addresses of other objects and are declared with `*` as follows:

  class-name`*` *identifier* `;`

```
int anInt = 123; // The int object is initialized
int* intPtr;     // intPtr stores an address
```

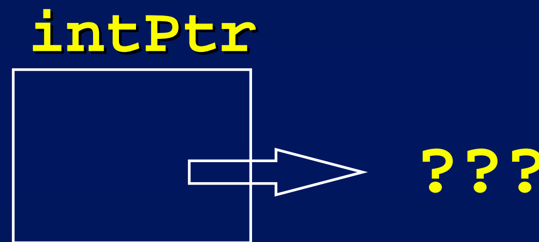
- `anInt` stores an integer, however, `intPtr` stores a pointer to an integer
- So pointer objects may store the address of other objects

The State of a Pointer Object

- At this point, the value of `intPtr` may have or become one of these values:
  - Undefined (as `intPtr` exists above)
  - The special null value 0,
    - after `intPtr = 0;`
  - The address of an int object,
    - after `intPtr = &anInt;`

The Value of intPtr

- Currently, we may depict the undefined value of **`intPtr`** as follows:

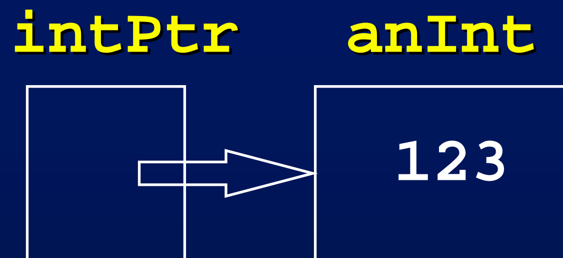


- The **`&`** symbol is called the *address-of operator* when it precedes an object.
- This assignment returns the address of **`anInt`** and stores that address into **`intPtr`**:

```
intPtr = &anInt;
```

Defining Pointer Objects

- The affect of this assignment
  `intPtr = &anInt;`
  is represented graphically like this:



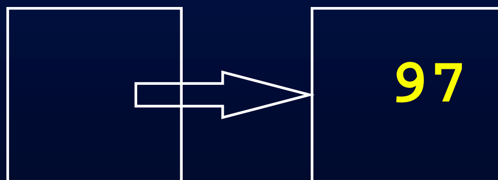
- Now `intPtr` is defined, but `anInt` is still undefined

Dereferencing

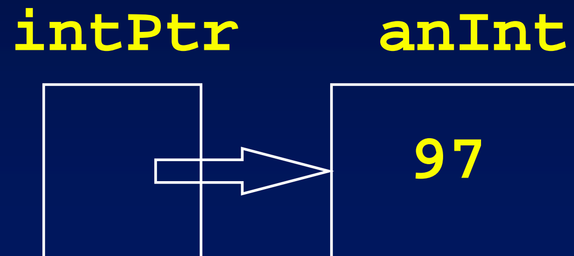
- We can define **anInt** with the usual assignment (**anInt = 0;**).
- Or we can initialize **anInt** indirectly with the dereference operator *****

```
anInt = 97;
intPtr* = 97;  // The same as anInt = 97
```

- Now both objects are defined:



The Dereference Operator



- The following code displays 97 and 96

```
cout << (intPtr*) << (intPtr*-1) << endl;
```

- and this code changes 97 to 98

```
intPtr*  = intPtr* + 1;
```

- Demonstrate pointer manipulation to indirectly swap two variables

Address-of and Dereference

- Write the output generated by this program

```
#include <iostream>
using namespace std;
int main()
{
  int *p1, *p2;
  int n1, n2;
  p1 = &n1;
  *p1 = 5;
  n2 = 10;
  p2 = &n2;
  cout << n1 << "  " << *p1 << endl;
  cout << n2 << "  " << *p2 << endl;
  return 0;
}
```

The Standard `list` *Class (from the STL)*

- A `list` object
  - stores a collection of objects
    - no need to determine maximum capacity at first
  - allows access to individual elements
    - either find one or iterate over all sequentially
- Here are some member functions

  `empty` Returns true if there are zero elements in the list

  `push_back` Adds one elements at end of the list

  `remove` Removes an object from the list if it is found

  `size` Returns the current number of objects in the container

Sample Program

```
// Demonstrate list, one of the standard container classes
#include <iostream>
#include <list>      // for the standard (STL) list class
using namespace std;

int main()
{
  list < int > intList;  // intList stores collection of ints

  cout << "intList.size() == " << intList.size() << endl;
```

Output: `intList.size() == 0`

```
  // "Push" 5 new int objects onto the "back" of the list.
  intList.push_back(111);
  intList.push_back(222);
  intList.push_back(333);
  intList.push_back(444);
  intList.push_back(555);
```

More sample messages

```
  // "Push" 5 new int objects onto the "back" of the list
  intList.push_back(111);
  intList.push_back(222);
  intList.push_back(333);
  intList.push_back(444);
  intList.push_back(555);
  // Remove an item in the list and
  // attempt to remove am object that is not
  intList.remove(333);       // remove 333
  intList.remove(999);       // not found
  // assert: There are four elements in the container intList.
  // assert: The first is 111 and the last is 555.
  // Show the number of objects in the container:
  cout << "intList.size() == " << intList.size() << endl;
  return 0;
}
```

Output: `intList.size() == 4`

Traversing a Standard vector Object

- Individual vector elements can be referenced
  - using subscript notation [ ]
  - using an iterator object
    - lists have iterator objects traverse the collection
- Another way to visit all vector elements
  - assume 5 vector elements are initialized like this:

```
vector<int> x(5);
x[0] = 11;
x[1] = 22;
x[2] = 33;
x[3] = 44;
x[4] = 55;
```

Using an iterator object

(same idea as iterator pattern)

```
// Construct an iterator object named i.
vector<int>::iterator i;
// i may indirectly reference any vector<int> element

cout << "*i   x[j]" << endl;
int j = 0;
// Now you don't need a variable 'n' to store the number
// of meaningful elements. Use end() member function instead
for(i = x.begin(); i != x.end(); i++) {
  cout << (*i) << "      " << x[j] << endl;
  // i stores an address, *i refers to the element
  // *i in this context is equivalent to x[j]
  j++;
}
```

Output

```
 *i    x[j]
 11     11
 22     22
 33     33
 44     44
 55     55
```

*The difference between i and *i*

- General form for constructing an iterator object

container-class **<** *class-of-elements* **> :: iterator** *object-name* **;**

- The *container-class* is any of the Standard Template Library (STL) Classes:

```
list   stack   queue   vector
```

- The *class-of-elements* is any class that has a default constructor and defines relational operators such as <
  - most standard classes can be contained
  - you have to overload operators for your own new classes
    - see chapter 18: Operator Overloading

Some Characteristics of the C++ list class

- A `list` object
  - manages a collection of objects
  - is sequenced—there is a first, second,...,last
  - can add objects at the beginning or at the end of the list with `push_front` and `push_back`
  - always knows how big it is
  - is generic because it may be declared to store collections of *any* class
  - may grow as big as computer memory allows
  - has many available algorithms available such as `find`, `remove`, `sort`

Search and Sort are built in

- The standard STL containers have algorithms defined for them. For example
  - sort: A member function to sort any collection
- This code builds a list of five string object

```
list<string> presidents;

presidents.push_back("George Washington");
presidents.push_back("John Adams");
presidents.push_back("Thomas Jefferson");
presidents.push_back("James Madison");
presidents.push_back("James Monroe");
```

Sort and Search the Easy way

- Then the list is sorted with the sort message

```
presidents.sort();
// assert: The list is in ascending order
```

- The list is searched from beginning to end for a specific object like this assuming == is defined

```
list<string>::iterator presPtr;
presPtr = find(presidents.begin(), presidents.end(),
                                     "Abraham Lincoln");
// prePtr is always == end if it wasn't found
if(presPtr != presidents.end())
  cout << "Found " << (*presPtr)  << endl;
else
  cout << "Did not find Abe" << endl;
```

#include <algorithm>

- sort is a member function
- find is a free function not defined vis

```
#include <list>
```

- To use the find algorithm

```
#include <algorithm>
```

- Question, write code that counts the number of times an element is found in a list