Chapter 12
Pointers and Memory Management

3rd Edition

Computing Fundamentals with C++

Rick Mercer

Franklin, Beedle & Associates

Goals

- Understand pointers store addresses of other objects
- Use primitive C++ arrays with no range checking
- Use several methods for initializing pointers
- Use the `new` and `delete` operators for memory management

Memory Considerations

- In addition to name, state, and operations, every object has an address, where the values are stored
- Objects also have a lifetime beginning with construction to when they are no longer accessible
- With the following initialization, we see that the name `charlie`, state `99`, and operations like `= + cout <<` are known

```
int charlie = 99; // But where is 99 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
- We can't predict addresses, but ints are four bytes so two integers could have addresses 4 bytes apart

```
int a = 123;
int b = 456
```

Address	Type	Name	State
6300	int	a	123
6304	int	b	456

Static and Dynamic Memory Allocation

- Some objects take a fixed amount of memory at compiletime:

  `char` `int` `double`

- Other objects require varying amounts of memory, which is allocated and deallocated dynamically, that is, at runtime, `string` for example
- We sometimes use pointers to allow for such *dynamic* objects

Pointers

- Pointer 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 value
- `intPtr` stores the address of variable
- So pointer objects may store the address of other objects

About Pointer Types

- Pointer objects store the address of other objects which are the same type as the type of the pointer
- An `int` pointer hold an addresses to a `int` object

```
int intP = 25;
int* intPtr = &intP;
```

- A `double` pointer hold an addresses to a `double`

```
double doubleD = 25.45;
double* doublePtr = &doubleD;
```

- A `Grid` pointer hold an addresses to a `Grid` object

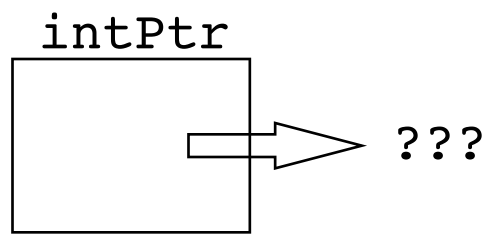
```
Grid gridG(5, 5, 0, 0, south);
Grid* gridPtr = &gridG;
```

The State of Pointers

- At this point, the value of `intPtr` may have or become one of these values
  - Undefined (as `intPtr` exists above)
  - The special value `nullptr` to indicate the pointer points to nothing: `intPtr = nullptr;`
  - The address of the `int` object: `intPtr=&anInt;`
    - `&` means address of

Pointer Values

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

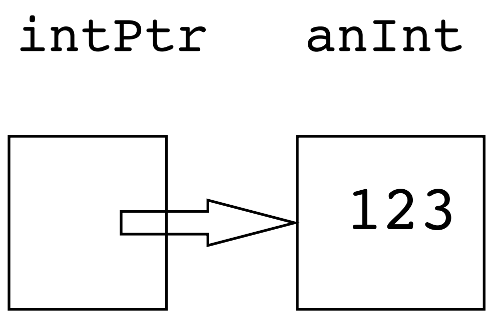


- The `&` symbol is called the *address-of operator* when it precedes an existing object
- This assignment returns the address of `anInt` and stores that address into `intPtr`:

```
intPtr = &anInt;
```

Defining Pointer Objects

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



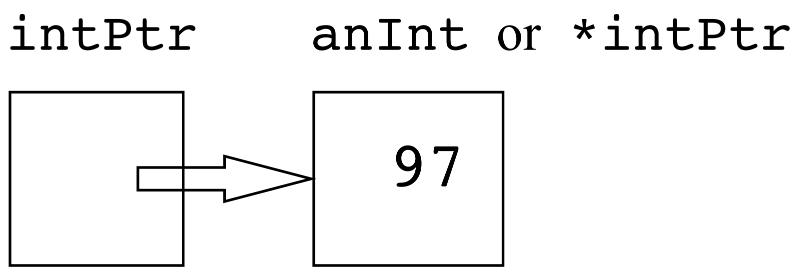
- Now `intPtr` is defined

The State of Pointers

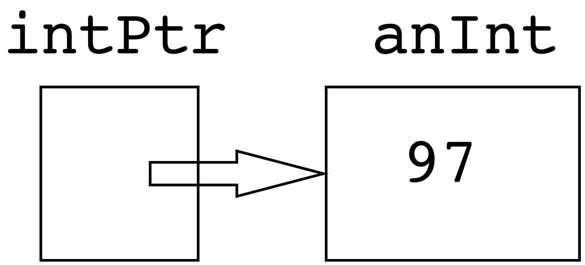
- We can change the value of `anInt` indirectly with the dereference operator `*`

```
*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;
```

- This code changes 97 to 98

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

The & operator

- The & operator has different meanings depending on how you use it.
  - When you use & to create a variable, you are creating a *reference*
  - When you use & in front of an existing variable the & is called the *address-of operator* and returns the address of the variable and not the value stored in the variable

The * operator

- The * operator also has different meanings depending on how you use it.
  - When you use * to create a variable, you are creating a pointer
  - When you use * in front of an existing pointer, you get the value stored at the address the pointer contains and not the address stored in the pointer
  - The * is also used in math operations when between numeric types

Address-of and Dereference

- What is 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;
}
```

Pointers to Objects

- Pointers can also store the addresses of objects with more than one value
- Because function calls have a higher precedence than dereferencing, override the priority scheme by wrapping the pointer dereference in parentheses

```
BankAccount anAcct("Ashley", 123.45);
BankAccount* bp;
bp = &anAcct;
(*bp).deposit(123.43);
cout << (*bp).getBalance(); // 246.88
```

Arrow Operator ->

- C++ also has an arrow operator to send message to object via its address (location in memory)

```
BankAccount anAcct("Ashley", 123.45);
BankAccount* bp;
bp = &anAcct;
bp->deposit(123.43);
cout << bp->getBalance(); // 246.88
```

The Primitive C Array

- C++ has primitive arrays

```
string myFriends[20]; // store up to 20 strings
double x[100];         // store up to 100 numbers
```

- There is no range checking with these

Compare C arrays to vector

Difference	vector Example	C Array Example
vectors can initialize all vector elements at construction; arrays cannot.	`vector <int> x(100, 0);` All elements are 0	`int x[100];` All garbage

vectors can be easily resized at runtime; arrays take a lot more work.	`int n;`
	`cin >> n;`
	`x.resize(n);`
		Can "grow" an array with more code

vectors can be made to prevent out-of-range subscripts.	You are told something is wrong `cin >> x.at(100);`	Destroys other memory `cin >> x[100];`
vectors require an `#include` primitive, built-in arrays do not.	`#include <vector>`	No `#include` required

The Array/Pointer Connection

- A primitive array is actually a pointer
  - The array name is actually the memory location of the very first array element
  - Individual array elements are referenced like this

  address of 1st array element + (subscript * size of 1 element)
- Arrays are automatically passed by reference when the parameter has `[ ]`

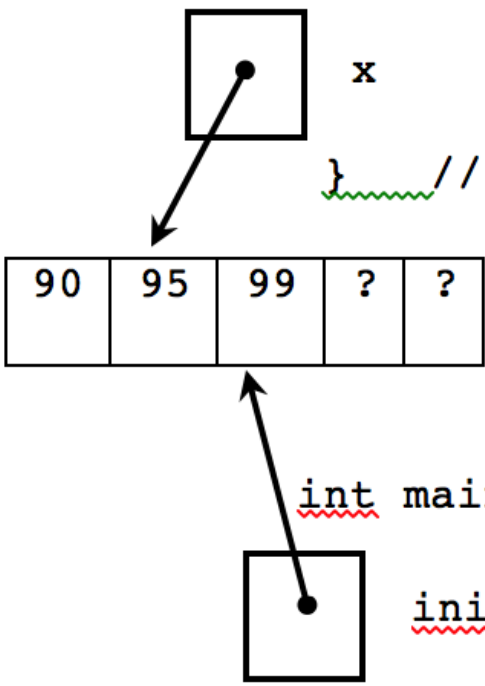
```
void init(int x[], int & n)
// Both x and n are reference parameters
```

Array parameters are reference parameters

- When passing arrays as parameters, you don't need `&`
  - `x` and `anArray` reference the same array object

```
void init(int x[], int & n) {

}   // A change to x is a change to anArray in main
```




```
int main() {

    init(anArray)

}   // A change to anArray is a change to x in init
```

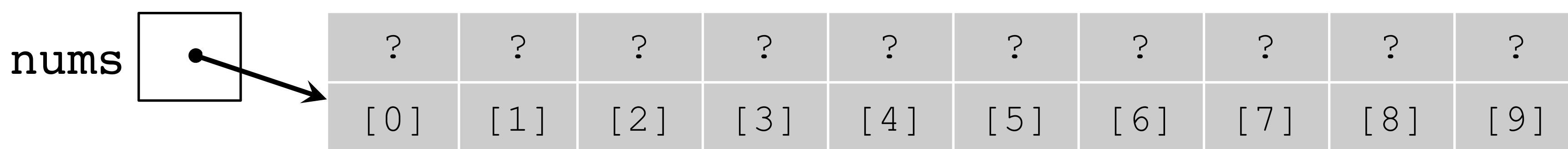
Allocating Memory with New

- Pointers can also be set with the C++ `new` operator
- This code allocates a contiguous block of memory to store the state.
  - It also returns the address, or a pointer to the object

```
int* intPtr = new int;
*intPtr = 123;
cout << *intPtr; // 123
```

- This code allocates a new array

```
int* nums = new int[10];
```



The `delete` Operator

- `new` allocates memory at runtime
- `delete` deallocates that memory to avoid memory leaks so it can be used by other new objects
- General form for recycling memory

```
delete pointer;
delete[] pointer-to-array;
```

- For the programs you write, you won't notice any difference by forgetting to delete
  - In a future course with destructors, or in an internship or job, you probably will