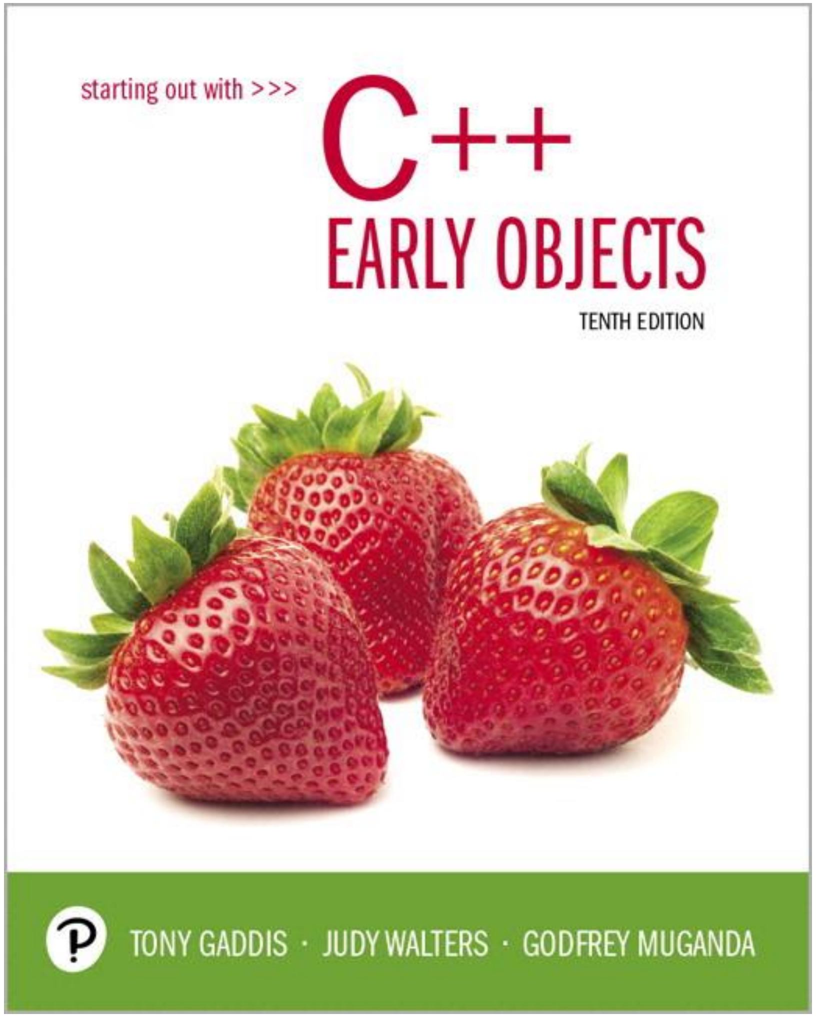
Starting Out with C++ Early Objects

Tenth Edition



Chapter 10

Pointers

Topics 1 of 2

10.1 Pointers and the Address Operator

10.2 Pointer Variables

10.3 The Relationship Between Arrays and Pointers

10.4 Pointer Arithmetic

10.5 Initializing Pointers

10.6 Comparing Pointers

10.7 Pointers as Function Parameters

Topics 2 of 2

10.8 Pointers to Constants and Constant Pointers

10.9 Dynamic Memory Allocation

10.10 Returning Pointers from Functions

10.11 Pointers to Class Objects and Structures

10.12 Selecting Members of Objects

10.13 Smart Pointers

10.1 Pointers and the Address Operator

- Each variable in a program is stored at a unique location in memory that has an address
- Use the address operator **&** to get the address of a variable:

```
int num = -23;
cout << &num; // prints address
              // in hexadecimal
```

- The address of a memory location is a pointer

Graphic 10-1

- Pg. 660

Hands-On Pg. 660

- Listing 10-1

10.2 Pointer Variables 1 of 3

- Pointer variable (pointer): a variable that holds an address
- Pointers provide an alternate way to access memory locations

Pointer Variables 2 of 3

- Definition:

```
int  *intptr;
```

- Read as:

  "**intptr** can hold the address of an int" or
  "**intptr** is a pointer to an int"

- The spacing in the definition does not matter:

```
int * intptr;
int*  intptr;
```

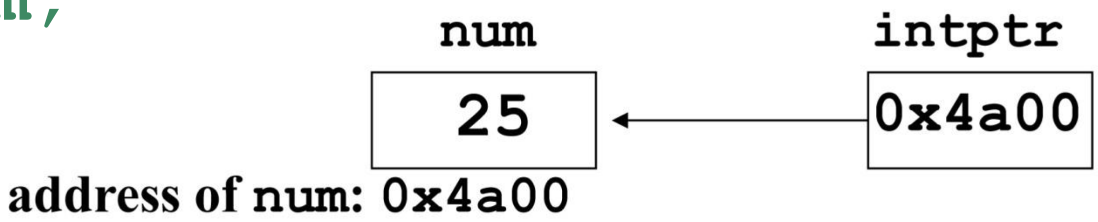
- ***** is called the indirection operator

Pointer Variables 3 of 3

- Definition and assignment:

```
int num = 25;
int *intptr;
intptr = &num;
```

- Memory layout:



- You can access **num** using **intptr** and indirection operator *****:

```
cout << intptr;  // prints 0x4a00
cout << *intptr; // prints 25
*intptr = 20;    // puts 20 in num
```

Hands-On Pg. 662

- Listing 662

Hands-On Pg. 663

- Listing 10-3

Hands-On Pg. 664

- Listing 10-4

10.3 The Relationship Between Arrays and Pointers 1 of 2

An array name is the starting address of the array

```
int vals[] = {4, 7, 11};
```

4	7	11

starting address of `vals`: `0x4a00`

```
cout << vals;     // displays 0x4a00
cout << vals[0];  // displays 4
```

The Relationship Between Arrays and Pointers 2 of 2

- An array name can be used as a pointer constant

```
int vals[] = {4, 7, 11};
cout << *vals;     // displays 4
```

- A pointer can be used as an array name

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

Hands-On Pg. 665

- Listing 10-5 (and graphic)

Pointers in Expressions

- Given:

```
int vals[]={4,7,11};
int *valptr = vals;
```

- What is **`valptr + 1`**?
- It means (address in **`valptr`**) + (1 * size of an **`int`**)

```
cout << *(valptr+1); // displays 7
cout << *(valptr+2); // displays 11
```

- Must use **`( )`** in expression

Pearson

Array Access 1 of 2

Array elements can be accessed in many ways

Array access method	Example
array name and `[ ]`	`vals[2] = 17;`
pointer to array and `[ ]`	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals+2) = 17;`
pointer to array and subscript arithmetic	`*(valptr+2) = 17;`

Array Access 2 of 2

- Array notation

```
vals[i]
```

is equivalent to the pointer notation

```
*(vals + i)
```

- Remember that no bounds checking is performed on array access. This applies to using a pointer for access as well as using the array name and a subscript.

Hands-On Pg. 666

- Listing 10-6

Hands-On Pg. 667

- Listing 10-7

Hands-On Pg. 668

- Listing 10-8

10.4 Pointer Arithmetic 1 of 5

Some arithmetic operators can be used with pointers:

- Increment and decrement operators **++**, **--**
- Integers can be added to or subtracted from pointers using the operators **+**, **-**, **+=**, and **-=**
- One pointer can be subtracted from another by using the subtraction operator **-**

Pointer Arithmetic 2 of 5

Assume the variable definitions

```
int vals[]={4,7,11};
int *valptr = vals;
```

Examples of use of **++** and **--**

```
valptr++; // points at 7
valptr--; // now points at 4
```

Pointer Arithmetic 3 of 5

Assume the variable definitions:

```
int vals[]={4,7,11};
int *valptr = vals;
```

Example of the use of **+** to add an int to a pointer:

```
cout << *(valptr + 2)
```

This statement will print 11

Pointer Arithmetic 4 of 5

Assume the variable definitions:

```
int vals[]={4,7,11};
int *valptr = vals;
```

Example of use of +=:

```
valptr = vals; // points at 4
valptr += 2;   // points at 11
```

Pointer Arithmetic 5 of 5

Assume the variable definitions

```
int vals[] = {4,7,11};
int *valptr = vals;
```

Example of pointer subtraction

```
valptr += 2;
cout << valptr - val;
```

This statement prints **2**: the number of **ints** between **valptr** and **val**

Hands-On Pg. 669

- Listing 10-9

10.5 Initializing Pointers

- You can initialize to NULL or 0 (zero)

```
int *ptr = NULL;
```

- You can initialize to addresses of other variables

```
int num, *numPtr = &num;
int val[ISIZE], *valptr = val;
```

- The initial value must have the correct type

```
float cost;
int *ptr = &cost; // won't work
```

Initializing Values in C++ 11

- In C++ 11, putting empty { } after a variable definition indicates that the variable should be initialized to its default value
- C++ 11 also has the the key word **`nullptr`** to indicate that a pointer variable does not contain a valid memory location
- You can use

```
int *ptr = nullptr;
```

- or

```
int *ptr{ };
```

10.6 Comparing Pointers

- Relational operators can be used to compare the addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses
if (*ptr1 == *ptr2) // compares
                    // contents
```

Hands-On Pg. 674

- Listing 10-10

10.7 Pointers as Function Parameters 1 of 3

- A pointer can be a parameter
- It works like a reference parameter to allow changes to argument from within a function
- A pointer parameter must be explicitly dereferenced to access the contents at that address

Pointers as Function Parameters 2 of 3

Requires:

1) asterisk * on parameter in prototype and header

```
void getNum(int *ptr);
```

2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```

3) address as argument to the function in the call

```
getNum(&num);
```

Pointers as Function Parameters 3 of 3

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int num1 = 2, num2 = -3;
swap(&num1, &num2);  //call
```

Hands-On Pg. 676

- Listing 10-11

Passing an Array Via a Pointer Parameter

- A pointer parameter receives an address when a function is called
- The address could be for a single variable, or it could be the address of the first element of an array.
- You can use either subscript notation or pointer arithmetic to access the elements of the array.

Hands-On Pg. 678

- Listing 10-12

10.8 Pointers to Constants and Constant Pointers

- A pointer to a constant: you cannot change the value that is pointed at
- A constant pointer: the address in the pointer cannot be changed after the pointer is initialized

Pearson

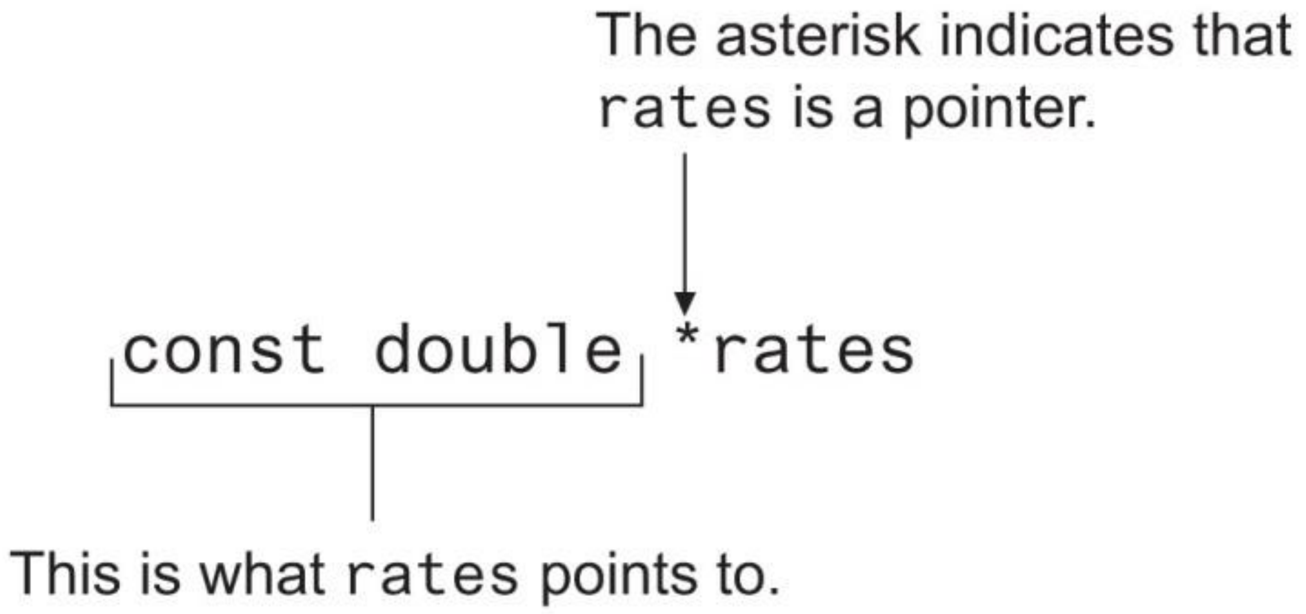
Pointers to Constants

- Must use the **`const`** keyword in the pointer definition:

```
const double taxRates[] =
                    {0.65, 0.8, 0.75};
const double *ratePtr;
```

- Use the **`const`** keyword for pointer parameters in function headers to protect data from modification in the function, as well as to pass addresses of **`const`** arguments

Pointer to Constant – What does the Definition Mean?



Read as: “rates is a pointer to a constant that is a double.”

Constant Pointers

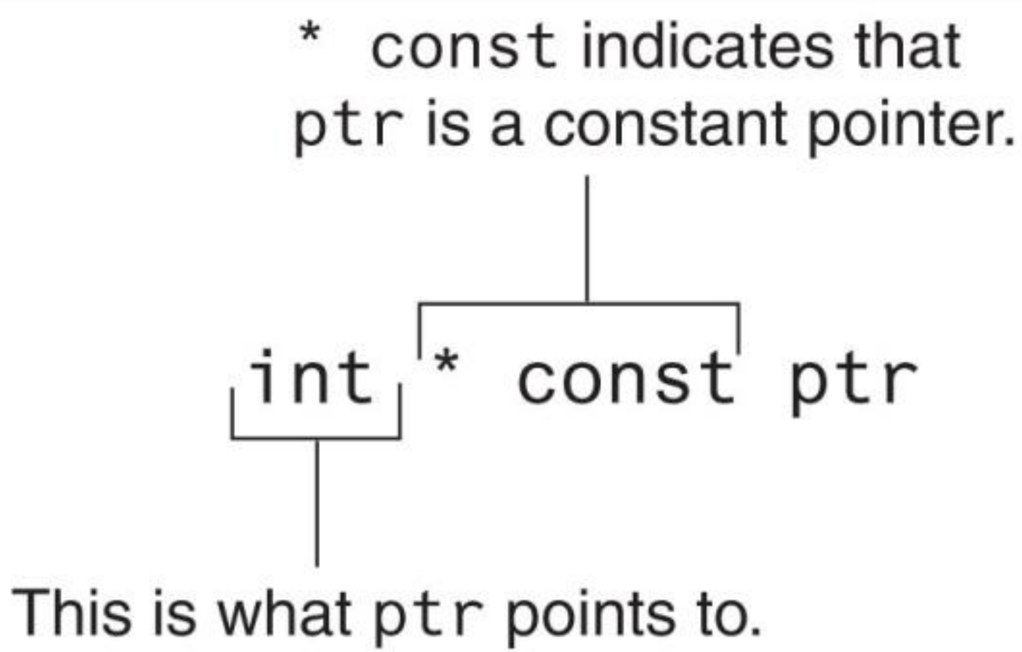
- A constant pointer is a pointer whose data (the address in the pointer) cannot change
- Defined with the **`const`** keyword next to the variable name:

```
int classSize = 24;
int * const classPtr = &classSize;
```

- It must be initialized when defined
- No initialization needed if used as a function parameter
  - Initialized by the argument when function is called
  - Arguments can differ on on different function calls
- While the address in the pointer cannot change, the data at that address may be changed

Pearson

Constant Pointer – What does the Definition Mean?



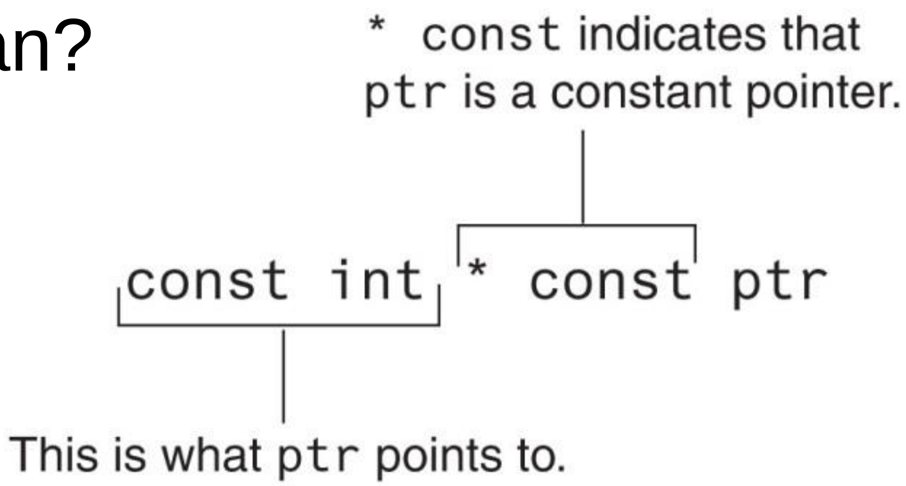
Read as: “ptr is a constant pointer to an int.”

Constant Pointer to Constant

- You can combine pointers to constants and constant pointers:

```
int size = 10;
const int * const ptr = &size;
```

- What does it mean?



Hands-On Pg. 681

- Listing 10-13

10.9 Dynamic Memory Allocation 1 of 2

- You can allocate storage for a variable while a program is running
- Use the **`new`** operator to allocate memory

```
double *dptr;
dptr = new double;
```

- **`new`** returns address of a memory location
- The data type of the variable is indicated after **`new`**

Dynamic Memory Allocation 2 of 2

- You can also use **`new`** to allocate an array

```
arrayPtr = new double[25];
```

  - The program may terminate if there is not sufficient memory

- You can then use **`[ ]`** or pointer arithmetic to access the array

Dynamic Memory Example

```
int *count, *arrayptr;
count = new int;
cout <<"How many students? ";
cin >> *count;
arrayptr = new int[*count];

for (int i=0; i<*count; i++)
{
  cout << "Enter score " << i << ": ";
  cin >> arrayptr[i];
}
```

Releasing Dynamic Memory

- Use **`delete`** to free dynamic memory

```
delete count;
```

- Use **`delete []`** to free dynamic array memory

```
delete [] arrayptr;
```

- Only use **`delete`** with dynamic memory!

Dangling Pointers and Memory Leaks

- A pointer is dangling if it contains the address of memory that has been freed by a call to **`delete`**.
  - Solution: set such pointers to NULL (or **`nullptr`** in C++ 11) as soon as the memory is freed.
- A memory leak occurs if no-longer-needed dynamic memory is not freed. The memory is unavailable for reuse within the program.
  - Solution: free up dynamic memory after use

Pearson

Hands-On Pg. 687

- Listing 10-14

10.10 Returning Pointers from Functions

- A pointer can be the return type of function

```
int* newNum();
```

- The function must not return a pointer to a local variable in the function
- The function should only return a pointer
  - to data that was passed to the function as an argument
  - to dynamically allocated memory

Hands-On Pg. 689

- Listing 10-15

More on Memory Leaks

General guidelines to avoid memory leaks:

- If a function allocates memory via **`new`**, it should, whenever possible, also deallocate the memory using **`delete`**
- If a class needs dynamic memory, it should
  - allocate it using **`new`** in the constructor
  - deallocate it using **`delete`** in the destructor

Hands-On Pg. 691

- Listing 10-16

10.11 Pointers to Class Objects and Structures

- You can create pointers to objects and structure variables

```
struct Student {…};
class Square {…};
Student stu1;
Student *stuPtr = &stu1;
Square sq1[4];
Square *squarePtr = &sq1[0];
```

- You need to use `()` when using `*` and `.` operators

```
(*stuPtr).studentID = 12204;
```

Structure Pointer Operator ->

- Simpler notation than **`(*ptr).member`**
- Use the form **`ptr->member`**

```
stuPtr->studentID = 12204;
squarePtr->setSide(14);
```

in place of the form **`(*ptr).member`**

```
(*stuPtr).studentID = 12204;
(*squarePtr).setSide(14);
```

Pearson

Dynamic Memory with Objects

- You can allocate dynamic structure variables and objects using pointers:

```
stuPtr = new Student;
```

- You can pass values to the object's constructor:

```
squarePtr = new Square(17);
```

- **`delete`** causes destructor to be invoked:

```
delete squarePtr;
```

Hands-On Pg. 695

- Listing 10-17

Structure/Object Pointers as Function Parameters

- Pointers to structures or objects can be passed as parameters to functions
- Such pointers provide a pass-by-reference parameter mechanism
- Pointers must be dereferenced in the function to access the member fields

Hands-On Pg. 696

- Listing 10-18

10.12 Selecting Members of Objects 1 of 2

Situation: A structure/object contains a pointer as a member. There is also a pointer *to* the structure/ object.

Problem: How do we access the pointer member via the structure/object pointer?

```
struct GradeList
  { string courseNum;
    int * grades;
  }
GradeList test1, *testPtr = &test1;
```

Pearson

Selecting Members of Objects 2 of 2

Expression	Meaning
`testPtr->grades`	Access the grades pointer in **`test1`**. This is the same as **`(*testPtr).grades`**
`*testPtr->grades`	Access the value pointed at by **`testPtr->grades`**. This is the same as **`*(*testPtr).grades`**
`*test1.grades`	Access the value pointed at by **`test1.grades`**

Pearson

10.13 Smart Pointers

- Introduced in C++ 11
- They can be used to solve the following problems in a large software project
  - dangling pointers – pointers whose memory is deleted while the pointer is still being used
  - memory leaks – allocated memory that is no longer needed but is not deleted
  - double-deletion – two different pointers de-allocating the same memory

Smart Pointers

- Smart pointers are objects that work like pointers.
- Unlike regular (raw) pointers, smart pointers can automatically delete dynamic memory that is no longer being used.
- There are three types of smart pointers:
  - unique pointers (**`unique_ptr`**)
  - shared pointers (**`shared_ptr`**)
  - weak pointers (**`weak_ptr`**)

Unique Pointers

- A smart pointer owns (or manages) the object that it points to.
- A unique pointer points to a dynamically allocated object that has a single owner.
- Ownership can be transferred to another unique pointer.
- Memory for the object is deallocated when the owning unique pointer goes out of scope, or if the unique pointer takes ownership of a different object.

Unique Pointer Examples

- Requires the `<memory>` header file
- Create a unique pointer that points to an int:

```
unique_ptr<int> uptr(new int);
```

- Assign the value 5 to it and print it:

```
*uptr = 5;
cout << *uptr;
```

- Transfer ownership to unique pointer ptr2:

```
unique_ptr<int> uptr2;
uptr2 = move(uptr);
```

Unique Pointers and Ownership Transfers – the `move()` function

- In a statement such as:

```
uptr2 = move(uptr);
```

  - Any object owned by **`uptr2`** is deallocated
  - **`uptr2`** takes ownership of the object previously owned by **`uptr`**
  - **`uptr`** becomes empty

The `move()` Function and Unique Pointers as Parameters

- The **`move()`** function is required on the argument when passing a unique pointer by value.
- The **`move()`** function is not required for pass by reference
- A unique pointer can be returned from a function, as the compiler automatically uses **`move()`** in this case.

Manually Clearing a Unique Pointer

- Unique pointers deallocate the memory for their objects when they go out of scope.
- To manually deallocate memory, use

```
uptr = nullptr;
```

or

```
uptr.reset();
```

Unique Pointers and Arrays

- Use array notation when using a unique pointer to allocate memory for an array

```
unique_ptr<int[]>uptr3(new int[5]);
```

- Doing so ensures that the proper deallocation (**`delete[]`** instead of **`delete`**) will be used.

Shared Pointers

- A smart pointer owns (or manages) the object that it points to.
- A shared pointer points to a dynamically allocated object that may have multiple owners.
- A control block manages the reference count of the number of shared owners and also possibly the raw pointer if one exists.

Creating Shared Pointers

- Create a shared pointer to point to an existing dynamic object declared with a raw pointer:

```
int * rawPtr = new int;
shared_ptr<int> uptr4(rawPtr);
```

- Create a second shared pointer initialized to the same object:

```
shared_ptr<int>uptr5 = uptr4;
```

- **`rawPtr`**, **`uptr4`**, and **`uptr5`** are all tracked in the control block.

How Many Control Blocks?

- Be careful that all references to a dynamic object are tracked in the same control block
- In the code below:

```
int * rawPtr = new int;
shared_ptr<int> uptr4(rawPtr);
shared_ptr<int> uptr5(rawPtr);
```

- Two control blocks are created. This can cause a dangling pointer.

Hands-On Pg. 704

- Listing 10-19

Memory Management Tip

- Creating a shared pointer involves memory for the object and memory for the control block.
- These memory allocations can be combined by using the **`make_shared`** function:

```
shared_ptr<int> uptr6 = make_shared<int>();
```

- You can also pass parameters to a constructor using an overloaded version of **`make_shared`**.

Hands-On Pg. 709, 710

- Listing 10-20
- Listing 10-22

Copyright