8.1 Why pointers?

A challenging and yet powerful programming construct is something called a *pointer*. A ***pointer*** is a variable that contains a memory address. This section describes a few situations where pointers are useful.

Vectors use dynamically allocated arrays

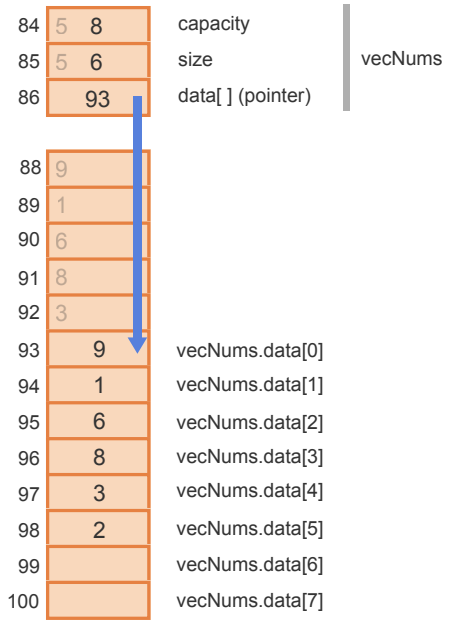
The C++ vector class is a container that internally uses a ***dynamically allocated array***, an array whose size can change during runtime. When a vector is created, the vector class internally dynamically allocates an array with an initial size, such as the size specified in the constructor. If the number of elements added to the vector exceeds the capacity of the current internal array, the vector class will dynamically allocate a new array with an increased size, and the contents of the array are copied into the new larger array. Each time the internal array is dynamically allocated, the array's location in memory will change. Thus, the vector class uses a pointer variable to store the memory location of the internal array.

The ability to dynamically change the size of a vector makes vectors more powerful than arrays. Built-in constructs have also made vectors safer to use in terms of memory management.

PARTICIPATION ACTIVITY 8.1.1: Dynamically allocated arrays.

```
vector<int> vecNums(5);
vecNums.at(0) = 9;
vecNums.at(1) = 1;
vecNums.at(2) = 6;
vecNums.at(3) = 8;
vecNums.at(4) = 3;
cout << "Size: " << vecNums.size() << endl;

vecNums.push_back(2);
cout << "New size: " << vecNums.size() << endl;
```

```
Size: 5
New size: 6
```



Animation content:

Static figure:
Begin Cpp code:

vector<int> vecNums(5);
vecNums.at(0) = 9;
vecNums.at(1) = 1;
vecNums.at(2) = 6;
vecNums.at(3) = 8;

```
vecNums.at(4) = 3;
cout << "Size: " << vecNums.size() << endl;

vecNums.push_back(2);
cout << "New size: " << vecNums.size() << endl;
```

End Cpp code.

Memory addresses 84 to 86 and 88 to 100 are shown. Variable, capacity, contains the value, 8, at memory address 84, variable, size, contains value, 6, at memory address 85, array pointer, data[], contains the value, 93, at memory address 86, and dynamically allocated array, vecNums, encapsulates all three memory addresses. An arrow points from the array pointer, data[], to the first memory address of the dynamically allocated array vecNums at memory address 93. Variable vecNums.data[0] contains the value 9 at memory address 93, variable vecNums.data[1] contains the value 1 at memory address 94, variable vecNums.data[2] contains the value 6 at memory address 95, variable vecNums.data[3] contains the value 8 at memory address 96, variable vecNums.data[4] contains the value 3 at memory address 97, variable vecNums.data[5] contains the value 2 at memory address 98, variable vecNums.data[6] contains no value at memory address 99, and variable vecNums.data[7] contains no value at memory address 100. The output box contains the lines, Size: 5 New size: 6.

Step 1: A vector internally uses a dynamically allocated array, an array whose size can change at runtime. To create a dynamically allocated array, a pointer stores the memory location of the array.

The line of code, vector<int> vecNums(5);, is higlighted and nine variables move from the line of code to the empty memory addresses 84 to 86 and 88 to 92. Variable capacity contains no value at memory address 84, variable size contains value 5 at memory address 85, array pointer, data[], contains the value 88 at memory address 86, and dynamically allocated array vecNums encapsulates all three memory addresses.

An arrow appears, pointing from the array pointer, data[], to the first memory address of the dynamically allocated array vecNums at memory address 88. Variable vecNums.data[0] contains no value at memory address 88, variable vecNums.data[1] contains no value at memory address 89, variable vecNums.data[2] contains no value at memory address 90, variable vecNums.data[3] contains no value at memory address 91, and variable vecNums.data[4] contains no value at memory address 92.

Step 2: A vector internally has data members for the size and capacity. Size is the current number of elements in the vector. capacity is the maximum number of elements that can be stored in the allocated array.

The line of code, vecNums.at(0) = 9;, is highlighted and the value 9 appears at variable vacNums.data[0] at memory address 88. The line of code, vecNums.at(1) = 1;, is highlighted and the value 1 appears at variable vacNums.data[1] at memory address 89. The line of code, vecNums.at(2) = 6;, is highlighted and the value 6 appears at variable vacNums.data[2] at memory address 90. The line of code, vecNums.at(3) = 8;, is highlighted and the value 8 appears at variable vacNums.data[3] at memory address 91. The line of code, vecNums.at(4) = 3;, is highlighted and the value 3 appears at variable vacNums.data[4] at memory address 92. The line of code, cout << "Size: " << vecNums.size() << endl;, is highlighted and the line, Size: 5, appears in the output box.

Step 3: push_back(2) needs to add a 6th element to the vector, but the capacity is only 5. push_back() allocates a new array with a larger capacity, copies existing elements to the new array, and adds the new element.

The line of code, vecNums.push_back(2);, is highlighted. Memory addresses 93 to 100 appear at the end of memory address 92. Variable newArray[0] contains no value at memory address 93, variable newArray[1] contains no value at memory address 94, variable newArray[2] contains no value at memory address 95, variable newArray[3] contains no value at memory address 96, variable newArray[4] contains no value at memory address 97, variable newArray[5] contains no value at memory address 98, variable newArray[6] contains no value at memory address 99, and variable newArray[7] contains no value at memory address 100. The values of memory adrresses 88 to 92 are copied to memory addresses 93 to 98. Variable newArray[0] contains the value 9, variable newArray[1] contains the value 1, variable newArray[2] contains the value 6, variable newArray[3] contains the value 8, variable newArray[4] contains the value 3. Finally, variable newArray[5] contains the new value, 2.

Step 4: Internally, the pointer for the vector's internal array is assigned to point to the new array, capacity is assigned with the new maximum size, size is incremented, and the previous array is freed.

The arrow from memory address 86, array pointer, data[], to memory address 88, vecNums.data[0], moves to point from memory address 86, array pointer, data[], to memory address 93 newArray[0]. The value, 88, in memory address 86, array pointer, data[], changes to 93. vecNums.data[0], vecNums.data[1], vecNums.data[2], vecNums.data[3], and vecNums.data[4] move from memory addresses 88 to 92 to memory adresses 93 to 97 replacing the newArray variables. Memory address 98 is renamed to vecNums.data[5], memory address 99 is renamed to vecNums.data[6] and memory address 100 is renamed to vecNums.data[7]. The values in memory addresses 88 to 92 fade away. The value, 5, in memory address 84, capacity, fades away and is assigned the value 8. The value, 5, in memory address 85, size, fades away and is assigned the value 6. The line of code, cout << "New size: " << vecNums.size() << endl;, is highlighted and the line, New size: 6, appears in the output box.

Animation captions:

1. A vector internally uses a dynamically allocated array, an array whose size can change at runtime. To create a dynamically allocated array, a pointer stores the memory location of the array.
2. A vector internally has data members for the size and capacity. Size is the current number of elements in the vector. capacity is the maximum number of elements that can be stored in the allocated array.
3. push_back(2) needs to add a 6th element to the vector, but the capacity is only 5. push_back() allocates a new array with a larger capacity, copies existing elements to the new array, and adds the new element.
4. Internally, the pointer for the vector's internal array is assigned to point to the new array, capacity is assigned with the new maximum size, size is incremented, and the previous array is freed.

PARTICIPATION ACTIVITY 8.1.2: Dynamically allocated arrays.

1) The size of a vector is the same as the vector's capacity.
   - ○ True
   - ○ False

2) When a dynamically allocated array increases capacity, the array's memory location remains the same.

- True
- False

3) Data that is stored in memory and no longer being used should be deleted to free up the memory.

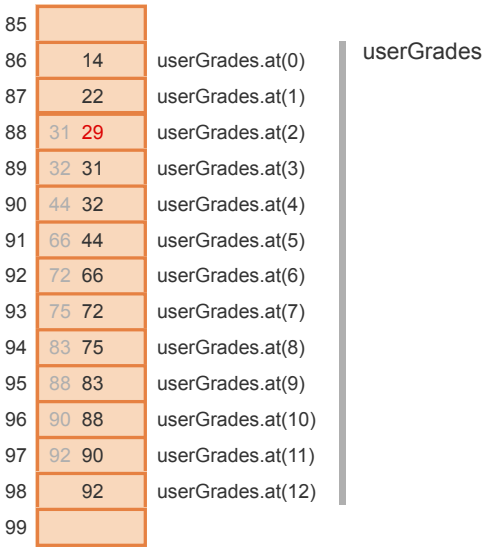
- True
- False

Inserting/erasing in vectors vs. linked lists

A vector (or array) stores a list of items in contiguous memory locations, which enables immediate access to any element of a vector userGrades by using `userGrades.at(i)` (or `userGrades[i]`). However, inserting an item requires making room by shifting higher-indexed items. Similarly, erasing an item requires shifting higher-indexed items to fill the gap. Shifting each item requires a few operations. If a program has a vector with thousands of elements, a single call to insert() or erase() can require thousands of instructions and cause the program to run very slowly, often called the ***vector insert/erase performance problem***.

PARTICIPATION ACTIVITY 8.1.3: Vector insert performance problem.

```
...
userGrades.insert(userGrades.begin() + 2, 29);
...
```



Animation content:

Static figure:
Begin Cpp code:

```
...
userGrades.insert(userGrades.begin() + 2, 29);
...
```

End Cpp code.

Memory addresses 85 to 99 are shown. Variable, userGrades.at(0), contains the value, 14, at memory address 86, variable, userGrades.at(1), contains the value, 22, at memory address 87, variable, userGrades.at(2), contains the value, 29, at memory address 88 variable, userGrades.at(3), contains the value, 31, at memory address 89 variable, userGrades.at(4), contains the value, 32, at memory address 90 variable, userGrades.at(5), contains the value, 44, at memory address 91 variable, userGrades.at(6), contains the value, 66, at memory address 92 variable, userGrades.at(7), contains the value, 72, at memory address 93 variable, userGrades.at(8), contains the value, 75, at memory address 94 variable, userGrades.at(9), contains the value, 83, at memory address 95 variable, userGrades.at(10), contains the value, 88, at memory address 96 variable, userGrades.at(11), contains the value, 90, at memory address 97 variable, userGrades.at(12), contains the value, 92, at memory address 98.

Step 1: Inserting an item at a specific location in a vector requires making room for the item by shifting higher-indexed items.

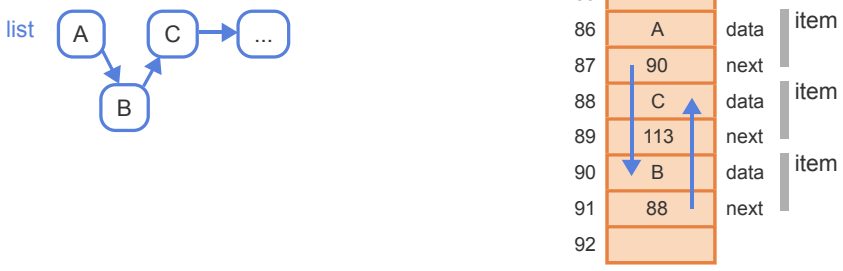
The line of code, userGrades.insert(userGrades.begin() + 2, 29);, is highlighted and the values from memory addresses 88 to 97 shift one memory address to memory addresses 89 to 98. The value, 29, is highlighted in the line of code, userGrades.insert(userGrades.begin() + 2, 29);, and moves to memory address 88.

Animation captions:

1. Inserting an item at a specific location in a vector requires making room for the item by shifting higher-indexed items.

A programmer can use a linked list to make inserts or erases faster. A ***linked list*** consists of items that contain both data and a pointer—a *link*—to the next list item. Thus, inserting a new item B between existing items A and C just requires changing A to point to B's memory location, and B to point to C's location, as shown in the following animation. No shifts occur.

PARTICIPATION ACTIVITY 8.1.4: A list avoids the shifting problem.



Animation content:

Static figure:

Memory addresses 85 to 92 are shown. Variable, data, contains the value, A, at memory address 86, variable, next, contains the value, 90, at memory address 87, variable, data, contains the value, C, at memory address 88, variable, next, contains the value, 113, at memory address 89, variable, data, contains the value, B, at memory address 90, and variable, data, contains the value, 88, at memory

address 91. Every consecutive data then next pair is considered an item. There is an arrow from memory address 87 to memory address 90 and from memory address 91 to memory address 88. There is a linked list with a node containing the value, A, pointing to a node containing the value, B. The node containing the value, B, is pointing to a node containing the value, C. The node containing the value, C, is pointing to a node containing the value, "...".

Step 1: List's first two items initially: (A, C, ...). Item A points to the next item at location 88. Item C points to next item at location 113 (not shown).

An arow points from memory address 87, containing the value 88, to memory address 88 containing the value, C.

Step 2: To insert new item B after item A, the new item B is first added to memory at location 90.

A new new containing the value, B, appears, the value B is assigned to memory adress 90, and the value 88 is assigned to memory address 91.

Step 3: Item B is set to point to location 88. Item A is updated to point to location 90. New list is (A, B, C, ...). No shifting of items after C was required.

An arrow appears pointing from the node containing the value, B, to the node containing the value, C. An arrow appears pointing from memory address 91 containing the value, 88, to memory address 88 containing the value, C. The arrow pointing from the node containing the value, A, to the node containing the value, C, changes to point from the node containing the value, A, to the node containing the value, B.

Animation captions:

1. List's first two items initially: (A, C, ...). Item A points to the next item at location 88. Item C points to next item at location 113 (not shown).
2. To insert new item B after item A, the new item B is first added to memory at location 90.
3. Item B is set to point to location 88. Item A is updated to point to location 90. New list is (A, B, C, ...). No shifting of items after C was required.

A vector is like people ordered by their seat in a theater row; if you want to insert yourself between two adjacent people, other people have to shift over to make room. A linked list is like people ordered by holding hands; if you want to insert yourself between two people, only those two people have to change hands, and nobody else is affected.

Table 8.1.1: Comparing vectors and linked lists.

Vector	Linked list

• Stores items in contiguous memory locations
• Supports quick access to i'th element via `at(i)`
  ◦ May be slow for inserts or erases on large lists due to necessary shifting of elements
	• Stores each item anywhere in memory, with each item pointing to the next list item
	• Supports fast inserts or deletes
	  ◦ access to i'th element may be slow as the list must be traversed from the first item to the i'th item

	• Uses more memory due to storing a link for each item

PARTICIPATION ACTIVITY 8.1.5: Inserting/erasing in vectors vs. linked lists.

For each operation, how many elements must be shifted? Assume no new memory needs to be allocated. Questions are for vectors, but also apply to arrays.

1) Append an item to the end of a 999-element vector (e.g., using push_back()).

Check Show answer

2) Insert an item at the front of a 999-element vector.

Check Show answer

3) Delete an item from the end of a 999-element vector.

Check Show answer

4) Delete an item from the front of a 999-element vector.

Check Show answer

5) Appending an item at the end of a 999-item linked list.

Check Show answer

6) Inserting a new item between the 10th and 11th items of a 999-item linked list.

Check Show answer

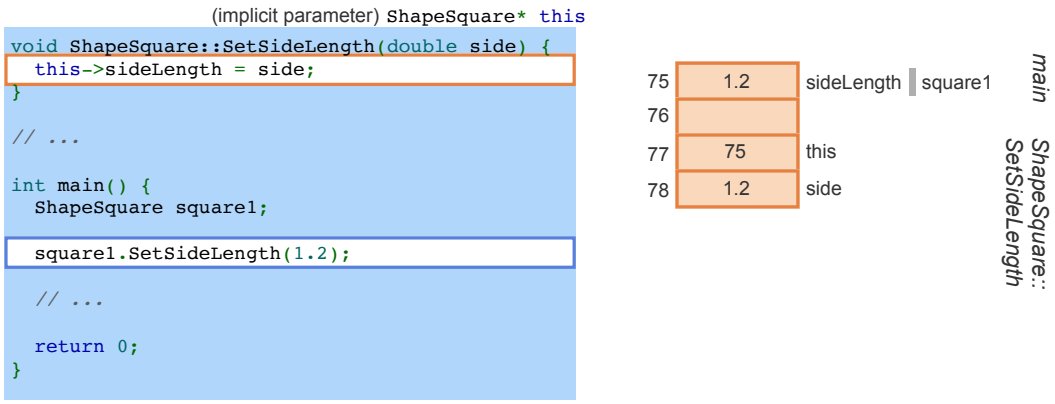
7) Finding the 500th item in a 999-item linked list requires visiting how many items? Correct answer is one of 0, 1, 500, and 999.

Check Show answer

Pointers used to call class member functions

When a class member function is called on an object, a pointer to the object is automatically passed to the member function as an implicit parameter called the ***this*** pointer. The `this` pointer enables access to an object's data members within the object's class member functions. A data member can be accessed using `this` and the member access operator for a pointer, `->`,ex. `this->sideLength`. The `this` pointer clearly indicates that an object's data member is being accessed, which is needed if a member function's parameter has the same variable name as the data member. The concept of the `this` pointer is explained further elsewhere.

PARTICIPATION ACTIVITY 8.1.6: Pointers used to call class member functions.



Animation content:

Static figure:
Begin Cpp code:

```
void ShapeSquare::SetSideLength(double side) {
  this->sideLength = side;
}
```

```
// ...

int main() {
  ShapeSquare square1;

  square1.SetSideLength(1.2);

  // ...

  return 0;
}
```


End Cpp code.

Memory addresses 75 to 78 are shown. The variable, sideLength, is at memory address 75, contains the value 1.2, and is labeled, square1. The variable, sideLength, is part of the memory labeled, main. The variable, this, is at memory address 77, and contains the value, 75. The variable, side, is at memory address 78 and contains the value 1.2. The variables, this and side, are part of the memory labeled, ShapeSquare::SetSideLength.

Step 1: square1 is a ShapeSquare object that has a double sideLength data member and a SetSideLength() member function.

The line of code, ShapeSquare square1;, is higlighted, the variable sideLength his assigned to memory address 75, is labeled , square1, and is part of the memory labeled main.

Step 2: Member functions have an implicit 'this' implicit parameter, which is a pointer to the class type. SetSideLength()'s implicit this parameter is a pointer to a ShapeSquare object.

The line of code, square1.SetSideLength(1.2);, is highlighted. The line of code, void ShapeSquare::SetSideLength(double side) {, is highlighted and the line, (implicit parameter) ShapeSquare* this, appears. Memory addresses 77 and 78 are lebeled ShapeSquare::SetSideLength. The variables, this and side, are assigned to memory address 77 and 78 respectively.

Step 3: When square1's SetSideLength() member function is called, square1's memory address is passed to the function using the 'this' implicit parameter.

The value, 75, of memory address 75 moves to memory address 77, and variable, this, is assigned to the value 75. The value 1.2 from the line of code, square1.SetSideLength(1.2);, and double side in the line of code, void ShapeSquare::SetSideLength(double side) {, moves to memory address 78, and vairable, side, is assigned with the value 1.2.

Step 4: The implicitly-passed square1 object pointer is clearly accessed within the member function via the name "this".

The value, 1.2, at memory address 78 duplicates, moves to memory address 75, and vairable sideLength is assigned with the value 1.2.

Animation captions:

1. square1 is a ShapeSquare object that has a double sideLength data member and a SetSideLength() member function.

2. Member functions have an implicit 'this' implicit parameter, which is a pointer to the class type. SetSideLength()'s implicit this parameter is a pointer to a ShapeSquare object.
3. When square1's SetSideLength() member function is called, square1's memory address is passed to the function using the 'this' implicit parameter.
4. The implicitly-passed square1 object pointer is clearly accessed within the member function via the name "this".

PARTICIPATION ACTIVITY 8.1.7: The 'this' pointer.

Assume the class FilmInfo has a private data member int filmLength and a member function `void SetFilmLength(int filmLength)`.

1) In SetFilmLength(), which would assign the data member filmLength with the value 120?
   - this->filmLength = 120;
   - this.filmLength = 120;
   - 120 = this->filmLength;

2) In SetFilmLength(), which would assign the data member filmLength with the parameter filmLength?
   - filmLength = filmLength;
   - this.filmLength = filmLength;
   - this->filmLength = filmLength;

Exploring further:

- Pointers tutorial from cplusplus.com
- Pointers article from cplusplus.com

8.2 Pointer basics

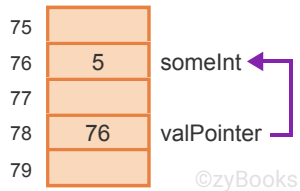
Pointer variables

A ***pointer*** is a variable that holds a memory address, rather than holding data like most variables. A pointer has a data type, and the data type determines what type of address is held in the pointer. Ex: An integer pointer holds a memory address of an integer, and a double pointer holds an address of a double. A pointer is declared by including * before the pointer's name. Ex: `int* maxItemPointer` declares an integer pointer named maxItemPointer.

Typically, a pointer is initialized with another variable's address. The ***reference operator*** (&) obtains a variable's address. Ex: `&someVar` returns the memory address of variable someVar. When a pointer is initialized with another variable's address, the pointer "points to" the variable.

PARTICIPATION ACTIVITY 8.2.1: Assigning a pointer with an address.

```
int main() {
   int someInt;
   int* valPointer;

   someInt = 5;
   cout << "someInt address is " << &someInt << endl;

   valPointer = &someInt;
   cout << "valPointer is " << valPointer << endl;

   return 0;
}
```




```
someInt address is 76
valPointer is 76
```

Animation content:

Static figure:
Begin Cpp code:

```
int main() {
  int someInt;
  int* valPointer;

  someInt = 5;
  cout << "someInt address is " << &someInt << endl;

  valPointer = &someInt;
  cout << "valPointer is " << valPointer << endl;

  return 0;
}
```

End Cpp code.

Memory addresses 75 to 79 are shown. The variable, someInt, is at memory address 76 and contains the value 5. The variable, valPointer, is at memory address 78 and contains the value 76. There is an arrow pointing from variable valPointer to variable someInt. The output console has the lines:
someInt address is 76
valPointer is 76

Step 1: someInt is located in memory at address 76.

The line of code, int someInt;, is highlighted and memory address 76 is labeled, someInt, with unknown value.

Step 2: valPointer is located in memory at address 78. valPointer has not been initialized, so valPointer points to an unknown address.

The line of code, int* valPointer;, is highlighted and memory address 78 is labeled, valPointer, with

unknown value.

Step 3: someInt is assigned with 5. The reference operator & returns someInt's address 76.

The line of code, someInt = 5;, is highlighted and memory address 76 is assigned with the value 5. The line of code, cout << "someInt address is " << &someInt << endl;, is highlighted and the line "someInt address is 76" is output to the console.

Step 4: valPointer is assigned with the memory address of someInt, so valPointer points to someInt.

The line of code, valPointer = &someInt;, is highlighted and the variable, valPointer, at memory address 78 is assigned the value 76. An arrow pointing from the variable, valPointer, at memory address 78 to the variable, someInt, at memory address 76 appears. The line of code, cout << "valPointer is " << valPointer << endl;, is highlighted and the line "valPointer is 76" is output to the console.

Animation captions:

1. someInt is located in memory at address 76.
2. valPointer is located in memory at address 78. valPointer has not been initialized, so valPointer points to an unknown address.
3. someInt is assigned with 5. The reference operator & returns someInt's address 76.
4. valPointer is assigned with the memory address of someInt, so valPointer points to someInt.

Printing memory addresses

The examples in this material show memory addresses using decimal numbers for simplicity. Outputting a memory address is likely to display a hexadecimal value like 006FF978 or 0x7ffc3ae4f0e4. Hexadecimal numbers are base 16, so the values use the digits 0-9 and letters A-F.

PARTICIPATION ACTIVITY 8.2.2: Declaring and initializing a pointer.

1) Declare a double pointer called sensorPointer.

Check Show answer

2) Output the address of the double variable sensorVal.

```
cout << ;
```

Check Show answer

3) Assign sensorPointer with the variable sensorVal's address. In other words, make sensorPointer point to sensorVal.

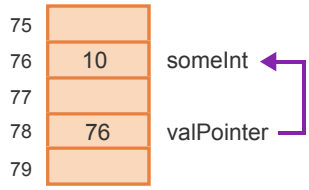
Check Show answer

Dereferencing a pointer

The ***dereference operator*** (*) is prepended to a pointer variable's name to retrieve the data to which the pointer variable points. Ex: If valPointer points to a memory address containing the integer 123, then `cout << *valPointer;` dereferences valPointer and outputs 123.

PARTICIPATION ACTIVITY 8.2.3: Using the dereference operator.

```
int main() {
   int someInt;
   int* valPointer;

   someInt = 5;
   cout << "someInt address is " << &someInt << endl;

   valPointer = &someInt;
   cout << "valPointer is " << valPointer << endl;

   cout << "*valPointer is " << *valPointer << endl;

   *valPointer = 10;   // Changes someInt to 10

   cout << "someInt is " << someInt << endl;
   cout << "*valPointer is " << *valPointer << endl;

   return 0;
}
```




```
someInt address is 76
valPointer is 76
*valPointer is 5
someInt is 10
*valPointer is 10
```

Animation content:

Static figure:
Begin Cpp code:

```
int main() {
  int someInt;
  int* valPointer;

  someInt = 5;
  cout << "someInt address is " << &someInt << endl;

  valPointer = &someInt;
  cout << "valPointer is " << valPointer << endl;

  cout << "*valPointer is " << *valPointer << endl;

  *valPointer = 10;   // Changes someInt to 10
```

```
   cout << "someInt is " << someInt << endl;
   cout << "*valPointer is " << *valPointer << endl;

   return 0;
}
```

End Cpp code.

Memory addresses 75 to 79 are shown. The variable, someInt, is at memory address 76 and contains the value 10. The variable, valPointer, is at memory address 78 and contains the value 76. There is an arrow pointing from variable valPointer to variable someInt. The output console has the lines:
someInt address is 76
valPointer is 76
*valPointer is 5
someInt is 10
*valPointer is 10

Step 1: someInt is located in memory at address 76, and valPointer points to someInt.

The lines of code,

```
int someInt;
int* valPointer;

someInt = 5;
cout << "someInt address is " << &someInt << endl;

valPointer = &someInt;
```

cout << "valPointer is " << valPointer << endl;, is highlighted and memory address 76 is labeled, someInt, with unknown value.,

are highlighted. Memory address 76 is labeled, someInt, and assigned the value 5. Memory address 78 is labeled, valPointer, and assigned the value 76. There is an arrow pointing from variable valPointer to variable someInt. The lines,

someInt address is 76
valPointer is 76,

are output to the console.

Step 2: The dereference operator * gets the value pointed to by valPointer, which is 5.

The line of code, cout << "*valPointer is " << *valPointer << endl;, is highlighted and the line "*valPointer is 5" is output to the console.

Step 3: Assigning *valPointer with a new value changes the value valPointer points to. The 5 changes to 10.

The line of code, *valPointer = 10; // Changes someInt to 10, is highlighted, and the variable, someInt, at memory address 76 is updated to the value 10.

Step 4: Changing *valPointer also changes someInt. someInt is now 10.

The line of code, cout << "someInt is " << someInt << endl;, is highlighted and the line "someInt is 10" is output to the console. The line of code, cout << "*valPointer is " << *valPointer << endl;, is highlighted and the line "*valPointer is 10" is output to the console.

Animation captions:

1. someInt is located in memory at address 76, and valPointer points to someInt.
2. The dereference operator * gets the value pointed to by valPointer, which is 5.
3. Assigning *valPointer with a new value changes the value valPointer points to. The 5 changes to 10.
4. Changing *valPointer also changes someInt. someInt is now 10.

PARTICIPATION ACTIVITY 8.2.4: Dereferencing a pointer.

Refer to the code below.

```
char userLetter = 'B';
char* letterPointer;
```

1) What line of code makes letterPointer point to userLetter?

- `letterPointer = userLetter;`
- `*letterPointer = &userLetter;`
- `letterPointer = &userLetter;`

2) What line of code assigns the variable outputLetter with the value letterPointer points to?

- `outputLetter = letterPointer;`
- `outputLetter = *letterPointer;`
- `someChar = &letterPointer;`

3) What does the code output?

```
letterPointer = &userLetter;
userLetter = 'A';
*letterPointer = 'C';
cout << userLetter;
```

- A
- B
- C

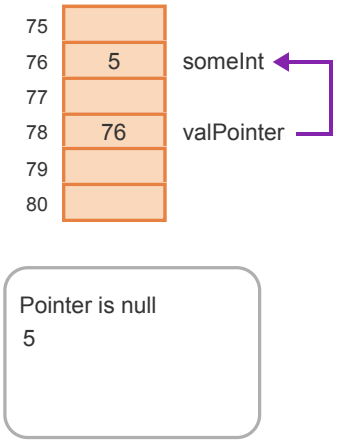
Null pointer

When a pointer is declared, the pointer variable holds an unknown address until the pointer is initialized. A programmer may wish to indicate that a pointer points to "nothing" by initializing a pointer to null. ***Null*** means "nothing". A pointer that is assigned with the keyword ***nullptr*** is said to be null. Ex: `int *maxValPointer = nullptr;` makes maxValPointer null.

In the animation below, the function PrintValue() only outputs the value pointed to by valuePointer if valuePointer is not null.

PARTICIPATION ACTIVITY 8.2.5: Checking to see if a pointer is null.


```
void PrintValue(int* valuePointer) {
   if (valuePointer == nullptr) {
      cout << "Pointer is null" << endl;
   }
   else {
      cout << *valuePointer << endl;
   }
}

int main() {
   int someInt = 5;
   int* valPointer = nullptr;

   PrintValue(valPointer);
   valPointer = &someInt;
   PrintValue(valPointer);

   return 0;
}
```



Animation content:

Static figure:
Begin Cpp code:

```
void PrintValue(int* valuePointer) {
  if (valuePointer == nullptr) {
    cout << "Pointer is null" << endl;
  }
  else {
    cout << *valuePointer << endl;
  }
}

int main() {
  int someInt = 5;
  int* valPointer = nullptr;

  PrintValue(valPointer);
  valPointer = &someInt;
  PrintValue(valPointer);

  return 0;
}
```

End Cpp code.

Memory addresses 75 to 80 are shown. The variable, someInt, is at memory address 76 and contains the value 5. The variable, valPointer, is at memory address 78 and contains the value 76. There is an arrow pointing from variable valPointer to variable someInt. The output console has the lines: Ponter is null
5

Step 1: someInt is located in memory at address 76. valPointer is assigned nullptr, so valPointer is null.

The line of code, int someInt = 5;, is highlighted and memory address 76 is labeled, someInt, and assigned the value 5. The line of code, int* valPointer = nullptr;, is highlighted and memory address 78 is labeled, valPointer, and assigned the value nullptr.

Step 2: valPointer is passed to PrintValue(), so the valuePointer parameter is assigned nullptr.

The line of code, PrintValue(valPointer);, is highlighted. The line of code, void PrintValue(int* valuePointer) {, is highlighted and memory address 80 is labeled, valuePointer, and assigned the value nullptr.

Step 3: The if statement is true since valuePointer is null.

The line of code, if (valuePointer == nullptr) {, is highlighted. The line of code, cout << "Pointer is null" << endl;, is highlighted and the line "Pointer is null" is output to the console.

Step 4: valPointer points to someInt, so calling PrintValue() assigns valuePointer with the address 76.

The line of code, valPointer = &someInt;, is highlighted. The variable, valPointer at memory address 78 is updated with the value 76. There is an arrow pointing from variable valPointer to variable someInt. The line of code, PrintValue(valPointer);, is highlighted. The line of code, void PrintValue(int* valuePointer) {, is highlighted and memory address 80 is labeled, valuePointer, and assigned the value 76. There is an arrow pointing from variable valuePointer to variable someInt.

Step 5: The if statement is false because valuePointer is no longer null. valuePointer points to the value 5, so 5 is output.

The line of code, if (valuePointer == nullptr) {, is highlighted. The line of code, cout << *valuePointer << endl;, is highlighted and the line "5" is output to the console.

Animation captions:

1. someInt is located in memory at address 76. valPointer is assigned nullptr, so valPointer is null.
2. valPointer is passed to PrintValue(), so the valuePointer parameter is assigned nullptr.
3. The if statement is true since valuePointer is null.
4. valPointer points to someInt, so calling PrintValue() assigns valuePointer with the address 76.
5. The if statement is false because valuePointer is no longer null. valuePointer points to the value 5, so 5 is output.

Null pointer

The nullptr keyword was added to the C++ language in version C++11. Before C++11, common practice was to use the literal 0 to indicate a null pointer. In C++'s predecessor language C, the macro NULL is used to indicate a null pointer.

PARTICIPATION ACTIVITY 8.2.6: Null pointer.

Refer to the animation above.

1) The code below outputs 3.

```
int numSides = 3;
int* valPointer = &numSides;
PrintValue(valPointer);
```

- True
- False

2) The code below outputs 5.

```
int numSides = 5;
int* valPointer = &numSides;
valPointer = nullptr;
PrintValue(valPointer);
```

- True
- False

3) The code below outputs 7.

```
int numSides = 7;
int* valPointer = nullptr;
cout << *valPointer;
```

- True
- False

Common pointer errors

A number of common pointer errors result in syntax errors that are caught by the compiler or runtime errors that may result in the program crashing.

Common syntax errors:

- *A common error is to use the dereference operator when initializing a pointer. Ex: For a variable declared* `int maxValue;` *and a pointer declared* `int* valPointer;`, `*valPointer = &maxValue;` *is a syntax error because *valPointer is referring to the value pointed to, not the pointer itself.*
- *A common error when declaring multiple pointers on the same line is to forget the * before each pointer name. Ex:* `int* valPointer1, valPointer2;` *declares valPointer1 as a pointer, but valPointer2 is declared as an integer because no * exists before valPointer2. Good practice is to declare one pointer per line to avoid accidentally declaring a pointer incorrectly.*

Common runtime errors:

- *A common error is to use the dereference operator when a pointer has not been initialized. Ex:* ***`cout << *valPointer;`*** *may cause a program to crash if valPointer holds an unknown address or an address the program is not allowed to access.*
- *A common error is to dereference a null pointer. Ex: If valPointer is null, then* ***`cout << *valPointer;`*** *causes the program to crash.* A pointer should always hold a valid address before the pointer is dereferenced.

PARTICIPATION ACTIVITY 8.2.7: Common pointer errors.

syntax errors

```
int someInt = 5;
int* valPointer;

*valPointer = &someInt;

valPointer = &someInt;
```

✖ int value cannot be assigned int*

◀ remove *

```
int* valPointer1, valPointer2;
valPointer1 = nullptr;
valPointer2 = nullptr;

int* valPointer1;
int* valPointer2;
```

✖ int cannot be assigned nullptr

◀ declare on separate lines

runtime errors

```
int* valPointer;
*valPointer = 4;

int someInt = 2;
valPointer = &someInt;
*valPointer = 4;
```

✖ dereferencing unknown address

◀ initialize pointer before dereferencing


```
int* valPointer = nullptr;
*valPointer = 4;

int someInt = 2;
valPointer = &someInt;
*valPointer = 4;
```

✖ dereferencing a null pointer

◀ initialize null pointer to valid address before dereferencing

Animation content:

Static figure:
Begin syntax error Cpp code 1:

int someInt = 5;
int* valPointer;

*valPointer = &someInt;

valPointer = &someInt;

End Cpp code.

Begin syntax error Cpp code 2:

valPointer1 = nullptr;
valPointer2 = nullptr;

```
int* valPointer1;
int* valPointer2;
```

End Cpp code.

Begin runtime error Cpp code 1:

```
int* valPointer;
*valPointer = 4;

int someInt = 2;
valPointer = &someInt;
*valPointer = 4
```


End Cpp code.

Begin runtime error Cpp code 2:

```
int* valPointer = nullptr;
*valPointer = 4;

int someInt = 2;
valPointer = &someInt;
*valPointer = 4;
```

End Cpp code.

Memory addresses 77 to 79 are shown for runtime error cpp code 1, and the variable, valPointer, is at memory address 78 and contains an unknown value. Memory addresses 77 to 79 are shown for runtime error cpp code 2, and the variable, valPointer, is at memory address 78 and contains an unknown value.

The line of code, *valPointer = &someInt;, in syntax error cpp code 1 is labeled "int value cannot be assigned int*" in red. The line of code, valPointer = &someInt;, in syntax error cpp code 1 is labeled "remove *" in green.

The line of code, valPointer2 = nullptr;, in syntax error cpp code 1 is labeled "" in red. The lines of code,

```
int* valPointer1;
int* valPointer2;,
```

in syntax error cpp code 1 is labeled "declare on separate lines" in green.

The line of code, *valPointer = 4;, in syntax error cpp code 1 is labeled "dereferencing unknown address" in red. The line of code,


```
int someInt = 2;
valPointer = &someInt;
*valPointer = 4;,
```

in syntax error cpp code 1 is labeled "initialize pointer before dereferencing" in green.

The line of code, *valPointer = 4;, in syntax error cpp code 1 is labeled "dereferencing a null pointer" in

red. The lines of code,

```
int someInt = 2;
valPointer = &someInt;
*valPointer = 4;,
```

in syntax error cpp code 1 is labeled "initialize null pointer to valid address before dereferencing" in green.

Step 1: someInt is located in memory at address 76.

The line of code, int someInt;, is highlighted and memory address 76 is labeled, someInt, with unknown value.

Step 2: valPointer is located in memory at address 78. valPointer has not been initialized, so valPointer points to an unknown address.

The line of code, int* valPointer;, is highlighted and memory address 78 is labeled, valPointer, with unknown value.

Step 3: someInt is assigned with 5. The reference operator & returns someInt's address 76.

The line of code, someInt = 5;, is highlighted and memory address 76 is assigned with the value 5. The line of code, cout << "someInt address is " << &someInt << endl;, is highlighted and the line "someInt address is 76" is output to the console.

Step 4: valPointer is assigned with the memory address of someInt, so valPointer points to someInt.

The line of code, valPointer = &someInt;, is highlighted and the variable, valPointer, at memory address 78 is assigned the value 76. An arrow pointing from the variable, valPointer, at memory address 78 to the variable, someInt, at memory address 76 appears. The line of code, cout << "valPointer is " << valPointer << endl;, is highlighted and the line "valPointer is 76" is output to the console.

Animation captions:

1. A syntax error results if valPointer is assigned using the dereference operator *.
2. Multiple pointers cannot be declared on a single line with only one asterisk. Good practice is to declare each pointer on a separate line.
3. valPointer is not initialized, so valPointer contains an unknown address. Dereferencing an unknown address may cause a runtime error.
4. valPointer is null, and dereferencing a null pointer causes a runtime error.

PARTICIPATION ACTIVITY 8.2.8: Common pointer errors.

Indicate if each code segment has a syntax error, runtime error, or no error.

1)

```
char* newPointer;
*newPointer = 'A';
cout << *newPointer;
```

- syntax error
- runtime error
- no errors

2)

```
char* valPointer1, *valPointer2;
valPointer1 = nullptr;
valPointer2 = nullptr;
```

- syntax error
- runtime error
- no errors

3)

```
char someChar = 'Z';
char* valPointer;
*valPointer = &someChar;
```

- syntax error
- runtime error
- no errors

4)

```
char* newPointer = nullptr;
char someChar = 'A';
*newPointer = 'B';
```

- syntax error
- runtime error
- no errors

Two pointer declaration styles

Some programmers prefer to place the asterisk next to the variable name when declaring a pointer. Ex: `int *valPointer;`. The style preference is useful when declaring multiple pointers on the same line: `int *valPointer1, *valPointer2;`. Good practice is to use the same pointer declaration style throughout the code: Either `int* valPointer` or `int *valPointer`.

This material uses the style `int* valPointer` and always declares one pointer per line to avoid accidentally declaring a pointer incorrectly.

Advanced compilers can check for common errors

Some compilers have advanced code analysis capabilities to catch some runtime errors at compile time. Ex: The compiler may issue a warning if the compiler detects a null pointer is being dereferenced. An advanced compiler can never catch all runtime errors because a potential runtime error may depend on user input, which is unknown at compile time.

zyDE 8.2.1: Using pointers.

The following provides an example (not useful other than for learning) of assigning the address of variable vehicleMpg to the pointer variable valPointer.

1. Run and observe that the two output statements produce the same output.
2. Modify the value assigned to *valPointer and run again.
3. Now uncomment the statement that assigns vehicleMpg. PREDICT whether both output statements will print the same output. Then run and observe the output. Did you predict correctly?

Load default template... Run

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     double vehicleMpg;
6     double* valPointer =
7
8     valPointer = &vehicle
9
10    *valPointer = 29.6;
11
12
13    // vehicleMpg = 40.0;
14
15    cout << "Vehicle MPG
```

CHALLENGE ACTIVITY 8.2.1: Enter the output of pointer content.

539740.3879454.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

int main() {
   int someNumber;
   int* numberPointer;

   someNumber = 1;
   numberPointer = &someNumber;

   cout << someNumber << " " << *numberPointer << endl;

   return 0;
}
```

```
1 1
```


1 2 3

Check Next

CHALLENGE ACTIVITY 8.2.2: Printing with pointers.

If the input is negative, make numItemsPointer be null. Otherwise, make numItemsPointer point to numItems and multiply the value to which numItemsPointer points by 10. Ex: If the user enters 99, the output should be:

```
Items: 990
```

Learn how our autograder works

539740.3879454.qx3zqy7

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int* numItemsPointer;
6     int numItems;
7
8     cin >> numItems;
9
10    /* Your solution goes here  */
11
12    if (numItemsPointer == nullptr) {
13       cout << "Items is negative" << endl;
14    }
15    else {
```

Run

CHALLENGE ACTIVITY 8.2.3: Pointer basics.

539740.3879454.qx3zqy7

Start

Given variables strength, speed, and direction, declare and assign the following pointers:

- integer pointer strengthPointer is assigned with the address of strength.

- double pointer speedPointer is assigned with the address of speed.
- character pointer directionPointer is assigned with the address of direction.

Ex: If the input is `4 39.5 E`, then the output is:

```
Hurricane level: 4
Speed: 39.5 miles per hour
Direction: E
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int strength;
   double speed;
   char direction;

   /* Your code goes here */

   cin >> strength;
   cin >> speed;
   cin >> direction;
```


1 2 3

Check Next level

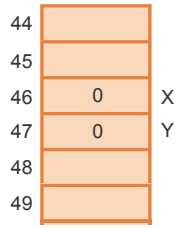
8.3 Operators: new, delete, and ->

The new operator

The ***new operator*** allocates memory for the given type and returns a pointer to the allocated memory. If the type is a class, the new operator calls the class's constructor after allocating memory for the class's member variables.

PARTICIPATION ACTIVITY 8.3.1: The new operator allocates space for an object, then calls the constructor.


```
#include <iostream>
using namespace std;

class Point {
public:
   Point();

   double X;
   double Y;
};

Point::Point() {
   cout << "In Point default constructor" << endl;
```



```
    X = 0;
    Y = 0;
}

int main() {
    Point* sample = new Point;
    cout << "Exiting main()" << endl;
    return 0;
}
```

Console:

```
In Point default constructor
Exiting main()
```


Animation content:

Static figure:
Begin Cpp code:

```
#include <iostream>
using namespace std;

class Point {
public:
  Point();

  double X;
  double Y;
};

Point::Point() {
  cout << "In Point default constructor" << endl;

  X = 0;
  Y = 0;
}

int main() {
  Point* sample = new Point;
  cout << "Exiting main()" << endl;
  return 0;
}
```

End Cpp code.

Memory addresses 44 to 51 are shown. The variable, X, is at memory address 46 and contains the value 0. The variable, Y, is at memory address 47 and contains the value 0. The variable, sample, is at memory address 51 and contains the value 46. The output console has the lines: In Point default constructor
Exiting main()

Step 1: The Point class contains two members, X and Y, both doubles.

The lines of code,

double X;
double Y;,

are highlighted.

Step 2: The new operator does 2 things. First, enough space is allocated for the Point object's 2 members, starting at memory address 46.

The code, new Point, and memory adresses 46 to 47 are highlighted

Step 3: Then the Point constructor is called, displaying a message and setting the X and Y values.

The line of code, cout << "In Point default constructor" << endl;, is highlighted. The line "In Point default constructor" is output to the console. The lines of code,

X = 0;
Y = 0;,

are highlighted. Variable X is assigned to memory adress 46 and contains the value 0. Variable Y is assigned to memory address 47 and contains the value 0. The line of code, }, is highlighted.

Step 4: The new operator returns a pointer to the allocated and initialized memory at address 46.

The line of code, Point* sample = new Point;, is highlighted. The line of code, cout << "Exiting main()" << endl;, is highlighted and the line "Exiting main()" is output to the console. The line of code, return 0;, is highlighted.

Animation captions:

1. The Point class contains two members, X and Y, both doubles.
2. The new operator does 2 things. First, enough space is allocated for the Point object's 2 members, starting at memory address 46.
3. Then the Point constructor is called, displaying a message and setting the X and Y values.
4. The new operator returns a pointer to the allocated and initialized memory at address 46.

PARTICIPATION ACTIVITY 8.3.2: The new operator.

1) The new operator returns an int.
   - True
   - False

2) When used with a class type, the new operator allocates memory after calling the class's constructor.
   - True
   - False

3) The new operator allocates, but does not deallocate, memory.

- True
- False

Constructor arguments

The new operator can pass arguments to the constructor. The arguments must be in parentheses following the class name.

PARTICIPATION ACTIVITY 8.3.3: Constructor arguments.

```
#include <iostream>
using namespace std;

class Point {
public:
   Point(double xValue = 0, double yValue = 0);
   void Print();

   double X;
   double Y;
};

Point:: Point(double xValue, double yValue) {
   X = xValue;
   Y = yValue;
}

void Point::Print() {
   cout << "(" << X << ", ";
   cout << Y << ")" << endl;
}

int main() {
   Point* point1 = new Point;
   (*point1).Print();

   Point* point2 = new Point(8, 9);
   (*point2).Print();

   return 0;
}
```

Console:

```
(0, 0)
(8, 9)
```

60	0	X
61	0	Y
62		
63	8	X
64	9	Y
65		
66		
67	60	point1
68		
69	63	point2
70		

Animation content:

Static figure:
Begin Cpp code:

#include <iostream>
using namespace std;

class Point {
public:
  Point(double xValue = 0, double yValue = 0);
  void Print();

  double X;
  double Y;
};

```
Point:: Point(double xValue, double yValue) {
   X = xValue;
   Y = yValue;
}

void Point::Print() {
   cout << "(" << X << ", ";
   cout << Y << ")" << endl;
}

int main() {
   Point* point1 = new Point;
   (*point1).Print();

   Point* point2 = new Point(8, 9);
   (*point2).Print();

   return 0;
}
```


End Cpp code.

Memory addresses 60 to 70 are shown. The variable, X, is at memory address 60 and contains the value 0. The variable, Y, is at memory address 61 and contains the value 0. Another variable, X, is at memory address 63 and contains the value 8. Another variable, Y, is at memory address 64 and contains the value 9. The variable, point1, is at memory address 67 and contains the value 60. The variable, point2, is at memory address 69 and contains the value 63. The output console has the lines: (0, 0)
(8, 9)

Step 1: The Point class contains 2 doubles, X and Y. The constructor has 2 parameters.

The lines of code,

```
Point(double xValue = 0, double yValue = 0);
double X;
double Y;,
```

are highlighted.

Step 2: "new Point" calls the constructor with no arguments. The default value of 0 is used for both numbers.

The line of code, Point* point1 = new Point;, is highlighted. The line of code, Point:: Point(double xValue, double yValue) {, is highlighted and the variables, xValue and yValue, both contain the value 0. The line of code, X = xValue;, is highlighted and the vairable, X, is assigned to memory address 60 and contains the value 0. The line of code, Y = yValue;, is highlighted and the vairable, Y, is assigned to memory address 61 and contains the value 0. The line of code, }, is highlighted.

Step 3: point1 is a pointer to the allocated object that resides at address 60. point1 is dereferenced, and the Print() member function is called.

The line of code, Point* point1 = new Point;, is highlighted and the vairable, point1, is assigned to memory address 67 and contains the value 60. The line of code, (*point1).Print();, is highlighted. The lines of code,

cout << "(" << X << ", ";
cout << Y << ")" << endl;,

are highlighted and the line "(0, 0)" is output to the console.

Step 4: "new Point(8, 9)" passes 8 and 9 as the constructor arguments.

The line of code, Point* point2 = new Point(8, 9);, is highlighted. The line of code, Point:: Point(double xValue, double yValue) {, is highlighted and the variables, xValue and yValue, contain the values 8 and 9 respectively. The line of code, X = xValue;, is highlighted and the vairable, X, is assigned to memory address 63 and contains the value 8. The line of code, Y = yValue;, is highlighted and the vairable, Y, is assigned to memory address 64 and contains the value 9. The line of code, }, is highlighted.

Step 5: point2 points to the object at address 63. Print() is called for point2.

The line of code, Point* point2 = new Point(8, 9);, is highlighted and the vairable, point2, is assigned to memory address 69 and contains the value 63. The line of code, (*point2).Print();, is highlighted. The lines of code,

cout << "(" << X << ", ";
cout << Y << ")" << endl;,

are highlighted and the line "(8, 9)" is output to the console.

Animation captions:

1. The Point class contains 2 doubles, X and Y. The constructor has 2 parameters.
2. "new Point" calls the constructor with no arguments. The default value of 0 is used for both numbers.
3. point1 is a pointer to the allocated object that resides at address 60. point1 is dereferenced, and the Print() member function is called.
4. "new Point(8, 9)" passes 8 and 9 as the constructor arguments.
5. point2 points to the object at address 63. Print() is called for point2.

PARTICIPATION ACTIVITY 8.3.4: Constructor arguments.

If unable to drag and drop, refresh the page.


```
Point* point = new Point(0, 0, 0);
```

```
Point* point = new Point(0, 10);
```

```
Point* point = new Point();
```

```
Point* point = new Point(10);
```

Constructs the point (0, 0).

	Constructs the point (10, 0).
	Constructs the point (0, 10).
	Causes a compiler error.

Reset

The member access operator

When using a pointer to an object, the ***member access operator*** (**->**) allows access to the object's members with the syntax `a->b` instead of `(*a).b`. Ex: If myPoint is a pointer to a Point object, `myPoint->Print()` calls the `Print()` member function.

Table 8.3.1: Using the member access operator.

Action	Syntax with dereferencing	Syntax with member access operator
Display point1's Y member value with cout	`cout << (*point1).Y;`	`cout << point1->Y;`
Call point2's Print() member function	`(*point2).Print();`	`point2->Print();`

PARTICIPATION ACTIVITY 8.3.5: The member access operator.

1) Which statement calls point1's Print() member function?

```
Point point1(20, 30);
```

- `(*point1).Print();`
- `point1->Print();`
- `point1.Print();`

2) Which statement calls point2's Print() member function?

```
Point* point2 = new Point(16,
8);
```

- `point2.Print();`
- `point2->Print();`

3) Which statement is *not* valid for multiplying point3's X and Y members?

```
Point* point3 = new Point(100,
50);
```

- `point3->X * point3->Y`
- `point3->X * (*point3).Y`
- `point3->X (*point3).Y`

The delete operator

The ***delete operator*** deallocates (or frees) a block of memory that was allocated with the new operator. The statement `delete pointerVariable;` deallocates a memory block pointed to by pointerVariable. If pointerVariable is null, delete has no effect.

After the delete, the program should not attempt to dereference pointerVariable since pointerVariable points to a memory location that is no longer allocated for use by pointerVariable. *Dereferencing a pointer whose memory has been deallocated is a common error and may cause strange program behavior that is difficult to debug. Ex: If pointerVariable points to deallocated memory that is later allocated to someVariable, changing *pointerVariable will mysteriously change someVariable.* Calling delete with a pointer that wasn't previously set by the new operator has undefined behavior and is a logic error.

PARTICIPATION ACTIVITY 8.3.6: The delete operator.

```
int main() {
   Point* point1 = new Point(73, 19);
   cout << "X = " << point1->X << endl;
   cout << "Y = " << point1->Y << endl;

   delete point1;

   // Error: can't use point1 after deletion
   point1->Print();
}
```

83	87	point1
84		
85		
86		
87	??	X
88	??	Y

Console:

```
X = 73
Y = 19
```

Animation content:

Static figure:
Begin Cpp code:

```
int main() {
  Point* point1 = new Point(73, 19);
  cout << "X = " << point1->X << endl;
  cout << "Y = " << point1->Y << endl;

  delete point1;

  // Error: can't use point1 after deletion
  point1->Print();
}
```

End Cpp code.

Memory addresses 83 to 88 are shown. The variable, point1, is at memory address 83 and contains the value 87. The variable, X, is at memory address 87 and contains an unknown value. The variable, Y, is at memory address 88 and contains an unknown value. The output console has the lines: X = 73 Y = 19

Step 1: point1 is allocated, and the X and Y members are displayed.

The line of code, Point* point1 = new Point(73, 19);, is highlighted. The vairable, point1, is assigned to memory address 83 and contains the value 87, the vairable, X, is assigned to memory address 87 and contains the value 73, and the vairable, Y, is assigned to memory address 88 and contains the value 19. The lines of code,

```
cout << "X = " << point1->X << endl;
cout << "Y = " << point1->Y << endl;
```

are highlighted and the lines,

```
X = 73
Y = 19
```

are output to the console.

Step 2: Deleting point1 frees the memory for the X and Y members. point1 still points to address 87.

The line of code, delete point1;, is highlighted and the values for variables X at memory address 87 and Y at memory address 88 become unknown.

Step 3: Since point1 points to deallocated memory, attempting to use point1 after deletion is a logic error.

The line of code, point1->Print();, is highlighted red.

Animation captions:

1. point1 is allocated, and the X and Y members are displayed.
2. Deleting point1 frees the memory for the X and Y members. point1 still points to address 87.
3. Since point1 points to deallocated memory, attempting to use point1 after deletion is a logic error.

PARTICIPATION ACTIVITY 8.3.7: The delete operator.

1) The delete operator can affect any pointer.

- True
- False

2) The statement `delete point1;` throws an exception if point1 is null.

- True
- False

3) After the statement `delete point1;` executes, point1 will be null.

- True
- False

Allocating and deleting object arrays

The new operator creates a dynamically allocated array of objects if the class name is followed by square brackets containing the array's length. A single, contiguous chunk of memory is allocated for the array, then the default constructor is called for each object in the array. A compiler error occurs if the class does not have a constructor that can take 0 arguments.

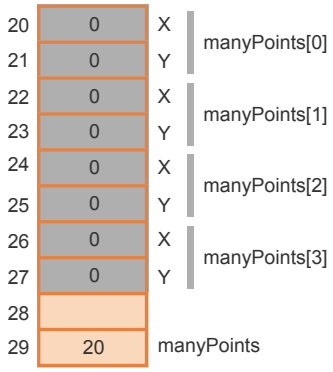
The ***delete[] operator*** is used to free an array allocated with the new operator.

PARTICIPATION ACTIVITY 8.3.8: Allocating and deleting an array of Point objects.

```
int main() {
   // Allocate points
   int pointCount = 4;
   Point* manyPoints = new Point[pointCount];

   // Display each point
   for (int i = 0; i < pointCount; ++i)
      manyPoints[i].Print();

   // Free all points with one delete
   delete[] manyPoints;

   return 0;
}
```

Console:

```
(0, 0)
(0, 0)
(0, 0)
(0, 0)
```



Animation content:

Static figure:
Begin Cpp code:


```
int main() {
  // Allocate points
  int pointCount = 4;
  Point* manyPoints = new Point[pointCount];

  // Display each point
  for (int i = 0; i < pointCount; ++i)
    manyPoints[i].Print();
```

```
  // Free all points with one delete
  delete[] manyPoints;

  return 0;
}
```

End Cpp code.

Memory addresses 20 to 29 are shown. The variable, manyPoints[0], encompasses variable X at memory address 20 and variable Y at memory address 21 which both contain the value 0. The variable, manyPoints[1], encompasses variable X at memory address 22 and variable Y at memory address 23 which both contain the value 0. The variable, manyPoints[2], encompasses variable X at memory address 24 and variable Y at memory address 25 which both contain the value 0. The variable, manyPoints[3], encompasses variable X at memory address 26 and variable Y at memory address 27 which both contain the value 0. The variable, manyPoints, is at memory address 29 and contains an the value 20. The output console has the lines: (0, 0)
(0, 0)
(0, 0)
(0, 0)

Step 1: point1 is allocated, and the X and Y members are displayed.

The lines of code,

```
int pointCount = 4;
Point* manyPoints = new Point[pointCount];,
```

. The variable, manyPoints[0], encompasses variable X at memory address 20 and variable Y at memory address 21 which both contain the value 0. The variable, manyPoints[1], encompasses variable X at memory address 22 and variable Y at memory address 23 which both contain the value 0. The variable, manyPoints[2], encompasses variable X at memory address 24 and variable Y at memory address 25 which both contain the value 0. The variable, manyPoints[3], encompasses variable X at memory address 26 and variable Y at memory address 27 which both contain the value 0. The variable, manyPoints, is at memory address 29 and contains an the value 20.

Step 2: Deleting point1 frees the memory for the X and Y members. point1 still points to address 87.

The lines of code,

```
for (int i = 0; i < pointCount; ++i)
  manyPoints[i].Print();
```


are highlighted. The line,

(0, 0)
(0, 0)
(0, 0)
(0, 0)

are output to the console.

Step 3: Since point1 points to deallocated memory, attempting to use point1 after deletion is a logic error.

The line of code, delete[] manyPoints;, is highlighted and the values for variable manyPoints at memory addresses 20 to 27 are grayed out. The line of code, return 0;, is highlighted.

Animation captions:

1. The new operator allocates a contiguous chunk of memory for an array of 4 Point objects. The default constructor is called for each, setting X and Y to 0.
2. Each point in the array is displayed.
3. The entire array is freed with the delete[] operator.

PARTICIPATION ACTIVITY 8.3.9: Allocating and deleting object arrays.

1) The array of points from the example above _____ contiguous in memory.

- might or might not be
- is always

2) What code properly frees the dynamically allocated array below?

```
Airplane* airplanes = new
Airplane[10];
```

- `delete airplanes;`
- `delete[] airplanes;`
-
```
for (int i = 0; i < 10;
++i) {
   delete airplanes[i];
}
```

3) The statement below only works if the Dalmatian class has _____.

```
Dalmatian* dogs = new
Dalmatian[101];
```

- no member functions
- only numerical member variables
- a constructor that can take 0 arguments

CHALLENGE ACTIVITY 8.3.1: Using the new, delete, and -> operators.

539740.3879454.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

class Car {
   public:
      Car(int distanceToSet);
   private:
      int distanceTraveled;
};

Car::Car(int distanceToSet) {
   distanceTraveled = distanceToSet;
   cout << "Traveled: " << distanceTraveled << endl;
}

int main() {
   Car* myCar1 = nullptr;
   Car* myCar2 = nullptr;

   myCar1 = new Car(75);
   myCar2 = new Car(65);

   return 0;
}
```

```
Traveled: 75
Traveled: 65
```


1 2 3

Check Next

CHALLENGE ACTIVITY 8.3.2: Deallocating memory

Deallocate memory for kitchenPaint using the delete operator. Note: Destructors, which use the "~" character, are explained in a later section.

Learn how our autograder works

539740.3879454.qx3zqy7

```
1  #include <iostream>
2  using namespace std;
3
4  class PaintContainer {
5     public:
6        ~PaintContainer();
7        double gallonPaint;
8  };
9
10 PaintContainer::~PaintContainer() { // Covered in section on Destructors.
11    cout << "PaintContainer deallocated." << endl;
12 }
13
14 int main() {
15    PaintContainer* kitchenPaint;
```

Run

CHALLENGE ACTIVITY 8.3.3: Operators: new, delete, and ->.

539740.3879454.qx3zqy7

Start

Two doubles are read as the base and the height of a Triangle object. Assign pointer myTriangle with a new Triangle object u the base and the height as arguments in that order.

Ex: If the input is `2.0 7.5`, then the output is:

```
Triangle's base: 2.0
Triangle's height: 7.5
```

```
#include <iostream>
#include <iomanip>
using namespace std;

class Triangle {
   public:
      Triangle(double baseValue, double heightValue);
      void Print();
   private:
      double base;
      double height;
};
Triangle::Triangle(double baseValue, double heightValue) {
   base = baseValue;
   height = heightValue;
```


1 2 3 4 5

Check Next level

Exploring further:

- operator new[] Reference Page from cplusplus.com
- More on operator new[] from msdn.microsoft.com
- operator delete[] Reference Page from cplusplus.com
- More on delete operator from msdn.microsoft.com
- More on -> operator from msdn.microsoft.com

8.4 String functions with pointers

C string library functions

The C string library, introduced elsewhere, contains several functions for working with C strings. This section describes the use of char pointers in such functions. The C string library must first be included via: `#include <cstring>`.

Each string library function operates on one or more strings, each passed as a char* or const char* argument. Strings passed as char* can be modified by the function, whereas strings passed as const char* arguments cannot. Examples of such functions are strcmp() and strcpy(), introduced elsewhere.

PARTICIPATION ACTIVITY

8.4.1: strcmp() and strcpy() string functions.

Function signatures:

```
int strcmp(const char* str1, const char* str2);

char* strcpy(char* destination, const char* source);
```


Usage example:

```
int cmp1, cmp2;
char string1[10] = "abcxyz";
char string2[10] = "xyz";
char newText[10];
char* subStr = nullptr;

cmp1 = strcmp(string1, string2);
cout << "strcmp of \"" << string1 << "\" and \"";
cout << string2 << "\" returned " << cmp1 << endl;

subStr = &string1[3];

cmp2 = strcmp(subStr, "xyz");
cout << "strcmp of \"" << subStr;
cout << "\" and \"xyz\" returned ";
cout << cmp2 << endl;

strcpy(newText, subStr);
cout << "newText is now \"" << newText << "\"";
cout << endl;
```

Console:

```
strcmp of "abcxyz" and "xyz" returned -23
strcmp of "xyz" and "xyz" returned 0
newText is now "xyz"
```

Animation content:

Static figure:
Begin Cpp code:

```
class MyClass {
public:
  MyClass();

  MyClass(int* someDataToCopy);


  ~MyClass();
  MyClass(const MyClass& objectToCopy);
  MyClass& operator=(const MyClass& objectToCopy);

  void MemberFunction1();
  void MemberFunction2();
  void MemberFunction3();
};
```


End Cpp code.

The line of code, MyClass();, is labeled Default constructor. The line of code, MyClass(int* someDataToCopy);, is labeled Other constructor (not a copy constructor). The lines of code,

~MyClass();
MyClass(const MyClass& objectToCopy);
MyClass& operator=(const MyClass& objectToCopy);,

are boxed and labeled The big three:
- Destructor
- Copy constructor
- Copy assignment operator.

Step 1: The big three consists of the destructor, copy constructor, and copy assignment operator.

The lines of code,

~MyClass();
MyClass(const MyClass& objectToCopy);
MyClass& operator=(const MyClass& objectToCopy);,

are boxed and labeled The big three:
- Destructor
- Copy constructor
- Copy assignment operator.

Step 2: The default constructor is not part of the big three.

The line of code, MyClass();, is labeled Default constructor.

Step 3: A constructor may exist that copies some data, but isn't the copy constructor. The copy constructor for MyClass takes a const MyClass& argument.

The line of code, MyClass(int* someDataToCopy);, is labeled Other constructor (not a copy constructor).

Animation captions:

1. strcmp() compares 2 strings. Since neither string is modified during the comparison, each parameter is a const char*.
2. strcmp() returns an integer that is 0 if the strings are equal, non-zero if the strings are not equal.
3. strcpy() copies a source string to a destination string. The destination string gets modified and thus is a char*.
4. The source string is not modified and thus is a const char*.
5. strcpy() copies 4 characters, "xyz" and the null-terminator, to newText.

PARTICIPATION ACTIVITY 8.4.2: C string library functions.

1) A variable declared as `char* substringAt5 = &myString[5];` cannot be passed as an argument to strcmp(), since strcmp() requires const char* arguments.
   - True
   - False

2) A character array variable declared as `char myString[50];` can be passed as either argument to strcpy().
   - True
   - False

3) A variable declared as `const char* firstMonth = "January";` could be passed as either argument to strcpy().
   - True
   - False

C string search functions

strchr(), strrchr(), and strstr() are C string library functions that search strings for an occurrence of a character or substring. Each function's first parameter is a const char*, representing the string to search within.

The strchr() and strrchr() functions find a character within a string, and thus have a char as the second parameter. strchr() finds the first occurrence of the character within the string and strrchr() finds the last occurrence.

strstr() searches for a substring within another string, and thus has a const char* as the second parameter.

Table 8.4.1: Some C string search functions.

Given:

```
char orgName[100] = "The Dept. of Redundancy Dept.";
char newText[100];
char* subString = nullptr;
```

strchr()	`strchr(sourceStr, searchChar)`
	
	Returns a null pointer if searchChar does not exist in sourceStr. Else, returns pointer to first occurrence.
		`if (strchr(orgName, 'D') != nullptr) {  // 'D' exists in orgName?`
		`   subString = strchr(orgName, 'D');    // Points to first 'D'`
		`   strcpy(newText, subString);          // newText now "Dept. of Redundancy Dept."`
		`}`
		
		`if (strchr(orgName, 'Z') != nullptr) {  // 'Z' exists in orgName?`
		`   ... // Doesn't exist, branch not taken`
		`}`

strrchr()	`strrchr(sourceStr, searchChar)`
	
	Returns a null pointer if searchChar does not exist in sourceStr. Else, returns pointer to LAST occurrence (searches in reverse, hence middle 'r' in name).
		`if (strrchr(orgName, 'D') != nullptr) {   // 'D' exists in orgName?`
		`   subString = strrchr(orgName, 'D');`
		`// Points to last 'D'`
		`   strcpy(newText, subString);            // newText now "Dept."`
		`}`

strstr()	`strstr(str1, str2)`
	
	Returns a null pointer if str2 does not exist in str1. Else, returns a char pointer pointing to the first character of the first occurrence of string str2 within string str1.
		`subString = strstr(orgName, "Dept"); // Points to first 'D'`
		`if (subString != nullptr) {`
		`   strcpy(newText, subString);       // newText now "Dept. of Redundancy Dept."`
		`}`

PARTICIPATION ACTIVITY 8.4.3: C string search functions.

1) What does fileExtension point to after the following code?

```
const char* fileName =
"Sample.file.name.txt";
const char* fileExtension =
strrchr(fileName, '.');
```

- ".file.name.txt"
- ".txt"
- "Sample.file.name"

2) `strstr(fileName, ".pdf")` is non-null only if the fileName string ends with ".pdf".

- True
- False

3) What is true about fileName if the following expression evaluates to true?

```
strchr(fileName, '.') ==
strrchr(fileName, '.')
```

- The '.' character occurs exactly once in fileName.
- The '.' character occurs 0 or 1 time in fileName.
- The '.' character occurs 1 or more times in fileName.

Search and replace example

The following example carries out a simple censoring program, replacing an exclamation point by a period and "Boo" by "---" (assuming those items are somehow bad and should be censored.) "Boo" is replaced using the strncpy() function, which is described elsewhere.

Note that only the first occurrence of "Boo" is replaced, as strstr() returns a pointer to the first occurrence. Additional code would be needed to delete all occurrences.

Figure 8.4.1: String searching example.

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
   const int MAX_USER_INPUT = 100;        // Max
input size
   char userInput[MAX_USER_INPUT];        // User
defined string
   char* stringPos = nullptr;             // Index
into string

   // Prompt user for input
   cout << "Enter a line of text: ";
   cin.getline(userInput, MAX_USER_INPUT);

   // Locate exclamation point, replace with period
   stringPos = strchr(userInput, '!');

   if (stringPos != nullptr) {
      *stringPos = '.';
   }

   // Locate "Boo" replace with "---"
   stringPos = strstr(userInput, "Boo");

   if (stringPos != nullptr) {
      strncpy(stringPos, "---", 3);
   }

   // Output modified string
   cout << "Censored: " << userInput << endl;

   return 0;
}
```

```
Enter a line of text:
Hello!
Censored: Hello.
...
Enter a line of text: Boo
hoo to you!
Censored: --- hoo to you.
...
Enter a line of text: Booo!
Boooo!!!!
Censored: ---o. Boooo!!!!
```


PARTICIPATION ACTIVITY 8.4.4: Modifying and searching strings.

1) Declare a char* variable named charPtr.

Check Show answer

2) Assuming `char* firstR;` is already declared, store in firstR a pointer to the first instance of an 'r' in the char* variable userInput.

Check Show answer

3) Assuming `char* lastR;` is already declared, store in lastR a pointer to the last instance of an 'r' in the char* variable userInput.

Check Show answer

4) Assuming `char* firstQuit;` is already declared, store in firstQuit a pointer to the first instance of "quit" in the char* variable userInput.

Check Show answer

CHALLENGE ACTIVITY 8.4.1: Enter the output of the string functions.

539740.3879454.qx3zqy7

Start

Type the program's output

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
   char nameAndTitle[50];
   char* stringPointer = nullptr;

   strcpy(nameAndTitle, "Dr. Allen Reed");

   stringPointer = strchr(nameAndTitle, 'N');
   if (stringPointer != nullptr) {
      cout << "a" << endl;
   }
   else {
      cout << "b" << endl;
   }

   return 0;
}
```

b

1 2 3 4 5

Check Next

CHALLENGE ACTIVITY 8.4.2: Find char in C string

Assign a pointer to any instance of searchChar in personName to searchResult.

Learn how our autograder works

539740.3879454.qx3zqy7

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
   char personName[100];
   char searchChar;
   char* searchResult = nullptr;

   cin.getline(personName, 100);
   cin >> searchChar;

   /* Your solution goes here  */

   if (searchResult != nullptr) {
```


Run

CHALLENGE ACTIVITY 8.4.3: Find C string in C string.

Assign the first instance of "The" in movieTitle to movieResult.

Learn how our autograder works

539740.3879454.qx3zqy7

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
   char movieTitle[100];
   char* movieResult = nullptr;

   cin.getline(movieTitle, 100);

   /* Your solution goes here  */

   cout << "Movie title contains The? ";
   if (movieResult != nullptr) {
      cout << "Yes." << endl;
```

Run

8.5 A first linked list

A common use of pointers is to create a list of items such that an item can be efficiently inserted somewhere in the middle of the list, without the shifting of later items as required for a vector. The following program illustrates how such a list can be created. A class is defined to represent each list item, known as a ***list node***. A node is comprised of the data to be stored in each list item, in this case just one int, and a pointer to the next node in the list. A special node named head is created to represent the front of the list, after which regular items can be inserted.

Figure 8.5.1: A basic example to introduce linked lists.

```
-1
555
999
777
```

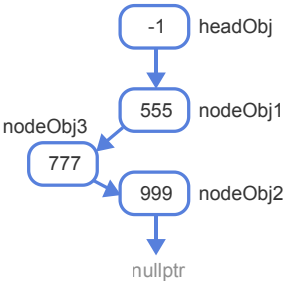

©zyBooks 01/31/24 17:50 1939727
Rob Daglio
MDCCOP2335Spring2024

```
#include <iostream>
using namespace std;

class IntNode {
public:
   IntNode(int dataInit = 0, IntNode* nextLoc =
nullptr);
   void InsertAfter(IntNode* nodeLoc);
   IntNode* GetNext();
   void PrintNodeData();
private:
   int dataVal;
   IntNode* nextNodePtr;
};

// Constructor
IntNode::IntNode(int dataInit, IntNode* nextLoc) {
   this->dataVal = dataInit;
   this->nextNodePtr = nextLoc;
}

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
void IntNode::InsertAfter(IntNode* nodeLoc) {
   IntNode* tmpNext = nullptr;

   tmpNext = this->nextNodePtr;    // Remember next
   this->nextNodePtr = nodeLoc;    // this -- node --
?
   nodeLoc->nextNodePtr = tmpNext; // this -- node --
next
}

// Print dataVal
void IntNode::PrintNodeData() {
   cout << this->dataVal << endl;
}

// Grab location pointed by nextNodePtr
IntNode* IntNode::GetNext() {
   return this->nextNodePtr;
}

int main() {
   IntNode* headObj  = nullptr; // Create IntNode
pointers
   IntNode* nodeObj1 = nullptr;
   IntNode* nodeObj2 = nullptr;
   IntNode* nodeObj3 = nullptr;
   IntNode* currObj  = nullptr;

   // Front of nodes list
   headObj = new IntNode(-1);

   // Insert nodes
   nodeObj1 = new IntNode(555);
   headObj->InsertAfter(nodeObj1);

   nodeObj2 = new IntNode(999);
   nodeObj1->InsertAfter(nodeObj2);

   nodeObj3 = new IntNode(777);
   nodeObj2->InsertAfter(nodeObj3);

   // Print linked list
   currObj = headObj;
   while (currObj != nullptr) {
      currObj->PrintNodeData();
      currObj = currObj->GetNext();
   }
```

```
   return 0;
}
```

PARTICIPATION ACTIVITY 8.5.1: Inserting nodes into a basic linked list.


```
headObj = new IntNode(-1);

// Add nodeObj1 after headObj
nodeObj1 = new IntNode(555);
headObj->InsertAfter(nodeObj1);

...

// Add nodeObj3 after nodeObj1
nodeObj1->InsertAfter(nodeObj3);
```



75	86	headObj	
76	84	nodeObj1	
77	82	nodeObj2	
78	80	nodeObj3	
79			
80	777	dataVal	*nodeOb
81	82	nextNodePtr	
82	999	dataVal	*nodeOb
83	nullptr	nextNodePtr	
84	555	dataVal	*nodeOb
85	80	nextNodePtr	
86	-1	dataVal	*headOb
87	84	nextNodePtr	

```
tmpNext = this->nextNodePtr;
this->nextNodePtr = nodeLoc;
nodeLoc->nextNodePtr = tmpNext;
```

Animation content:

Static figure: Two code blocks, a linked list, and a column of memory locations.
Begin Cpp code:
headObj = new IntNode(-1);

// Add nodeObj1 after headObj
nodeObj1 = new IntNode(555);
headObj->InsertAfter(nodeObj1);
...
// Add nodeObj3 after nodeObj1
nodeObj1->InsertAfter(nodeObj3);
End Cpp code.

Begin Cpp code:
tmpNext = this->nextNodePtr;
this->nextNodePtr = nodeLoc;
nodeLoc->nextNodePtr = tmpNext;
End Cpp code.

The program in the static figure is executed and the steps and substeps of the program are described in the following five tables. The 13 memory locations, addresses 75 to 87, are initially empty.

Step 1: The headObj pointer points to a special node that represents the front of the list. When the list

is first created, no list items exist, so the head node's nextNodePtr pointer is null.

Code executed	Memory address and label	Value stored	Additional information
headObj = new IntNode(-1);	86: dataVal	-1	Memory locations 86 and 87 are labeled *headObj. A linked list appears with the first node -1, labeled headObj, and an arrow heading down to the word nullptr.
	87: nextNodePtr	nullptr	
	75: headObj	86	

Step 2: To insert a node in the list, the new node nodeObj1 is first created with the value 555.

Code executed	Memory address and label	Value stored	Additional information
nodeObj1 = new IntNode(555);	84: dataVal	555	Memory locations 84 and 85 are labeled *nodeObj1. Another node 555, labeled nodeObj1, and an arrow heading down to the word nullptr appears.
	85: nextNodePtr	nullptr	
	76: nodeObj1	84	

Step 3: To insert the new node, tmpNext is pointed to the head node's next node, the head node's nextNodePtr is pointed to the new node, and the new node's nextNodePtr is pointed to tmpNext.

Code executed	Memory address and label	Value stored	Additional information
headObj->InsertAfter(nodeObj1); tmpNext = this->nextNodePtr;	79: tmpNext	nullptr	
this->nextNodePtr = nodeLoc;	87: nextNodePtr	84	In the linked list, node -1 now points to node 555. Node 555 still points to nullptr.
nodeLoc->nextNodePtr = tmpNext;	79: tmpNext		nullptr from tmpNext slides down to be the nullptr in nextNodePtr
	85: nextNodePtr	nullptr	

Step 4: A second node nodeObj2 with the value 999 is inserted at the end of the list, and a third node nodeObj3 with the value 777 is created.

Code executed	Memory address and label	Value stored	Additional information
headObj->InsertAfter(nodeObj1); ...	77: nodeObj2	82	The memory locations are filled in. Memory locations 82 and 83 are labeled *nodeObj2, and 80 and 81 are labeled *nodeObj3. In the linked list, node -1, labeled headObj, points to node 555. Node 555, labeled nodeObj1 points to node 999. Node 999, labeled nodeObj2, points to nullptr. Node 777, labeled Obj3, points to nullptr.
	78: nodeObj3	80	
	80: dataVal	777	
	81: nextNodePtr	nullptr	
	82: dataVal	999	

	83: nextNodePtr	nullptr	
	85: nextNodePtr	82	

Step 5: To insert nodeObj3 after nodeObj1, tmpNext is pointed to the nodeObj1's next node, the nodeObj1's nextNodePtr is pointed to the nodeObj3, and nodeObj3's nextNodePtr is pointed to tmpNext.

Code executed	Memory address and label	Value stored	Additional information
nodeObj1->InsertAfter(nodeObj3); tmpNext = this->nextNodePtr;	79: tmpNext	82	
this->nextNodePtr = nodeLoc;	85: nextNodePtr	80	In the linked list, node 555 now points to node 777.
nodeLoc->nextNodePtr = tmpNext;	79: tmpNext		In the linked list, node 777 now point to node 999, and node 999 still points to nullptr.
	81: nextNodePtr	82	

Animation captions:

1. The headObj pointer points to a special node that represents the front of the list. When the list is first created, no list items exist, so the head node's nextNodePtr pointer is null.
2. To insert a node in the list, the new node nodeObj1 is first created with the value 555.
3. To insert the new node, tmpNext is pointed to the head node's next node, the head node's nextNodePtr is pointed to the new node, and the new node's nextNodePtr is pointed to tmpNext.
4. A second node nodeObj2 with the value 999 is inserted at the end of the list, and a third node nodeObj3 with the value 777 is created.
5. To insert nodeObj3 after nodeObj1, tmpNext is pointed to the nodeObj1's next node, the nodeObj1's nextNodePtr is pointed to the nodeObj3, and nodeObj3's nextNodePtr is pointed to tmpNext.

The most interesting part of the above program is the InsertAfter() function, which inserts a new node after a given node already in the list. The above animation illustrates.

PARTICIPATION ACTIVITY 8.5.2: A first linked list.

Some questions refer to the above linked list code and animation.

1) A linked list has what key advantage over a sequential storage approach like an array or vector?

- An item can be inserted somewhere in the middle of the list without having to shift all subsequent items.
- Uses less memory overall.
- Can store items other than int variables.

2) What is the purpose of a list's head node?

- Stores the first item in the list.
- Provides a pointer to the first item's node in the list, if such an item exists.
- Stores all the data of the list.

3) After the above list is done having items inserted, at what memory address is the last list item's node located?

- 80
- 82
- 84
- 86

4) After the above list has items inserted as above, if a fourth item was inserted at the front of the list, what would happen to the location of node1?

- Changes from 84 to 86.
- Changes from 84 to 82.
- Stays at 84.

In contrast to the above program that declares one variable for each item allocated by the new operator, a program commonly declares just one or a few variables to manage a large number of items allocated using the new operator. The following example replaces the above main() function, showing how just two pointer variables, currObj and lastObj, can manage 20 allocated items in the list.

To run the following figure, `#include <cstdlib>` was added to access the rand() function.

Figure 8.5.2: Managing many new items using just a few pointer variables.

```
#include <iostream>
#include <cstdlib>
using namespace std;

class IntNode {
public:
   IntNode(int dataInit = 0, IntNode* nextLoc = nullptr);
   void InsertAfter(IntNode* nodeLoc);
   IntNode* GetNext();
   void PrintNodeData();
private:
   int dataVal;
   IntNode* nextNodePtr;
};

// Constructor
IntNode::IntNode(int dataInit, IntNode* nextLoc) {
   this->dataVal = dataInit;
   this->nextNodePtr = nextLoc;
}

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
void IntNode::InsertAfter(IntNode* nodeLoc) {
   IntNode* tmpNext = nullptr;

   tmpNext = this->nextNodePtr;     // Remember next
   this->nextNodePtr = nodeLoc;     // this -- node -- ?
   nodeLoc->nextNodePtr = tmpNext; // this -- node -- next
}

// Print dataVal
void IntNode::PrintNodeData() {
   cout << this->dataVal << endl;
}

// Grab location pointed by nextNodePtr
IntNode* IntNode::GetNext() {
   return this->nextNodePtr;
}

int main() {
   IntNode* headObj = nullptr; // Create IntNode pointers
   IntNode* currObj = nullptr;
   IntNode* lastObj = nullptr;
   int i;                // Loop index

   headObj = new IntNode(-1);        // Front of nodes list
   lastObj = headObj;

   for (i = 0; i < 20; ++i) {        // Append 20 rand nums
      currObj = new IntNode(rand());

      lastObj->InsertAfter(currObj); // Append curr
      lastObj = currObj;             // Curr is the new last
item
   }

   currObj = headObj;                // Print the list

   while (currObj != nullptr) {
      currObj->PrintNodeData();
      currObj = currObj->GetNext();
   }

   return 0;
}
```

```
-1
16807
282475249
1622650073
984943658
1144108930
470211272
101027544
1457850878
1458777923
2007237709
823564440
1115438165
1784484492
74243042
114807987
1137522503
1441282327
16531729
823378840
143542612
```


©zyBooks 01/31/24 17:50 1939727
Rob Daglio
MDCCOP2335Spring2024

zyDE 8.5.1: Managing a linked list.

Finish the program so that it finds and prints the smallest value in the linked list.

Load default template... Run


```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  class IntNode {
6  public:
7     IntNode(int dataInit
8     void InsertAfter(IntN
9     IntNode* GetNext();
10    void PrintNodeData();
11    int GetDataVal();
12 private:
13    int dataVal;
14    IntNode* nextNodePtr;
15 };
```

Normally, a linked list would be maintained by member functions of another class, such as IntList. Private data members of that class might include the list head (a list node allocated by the list class constructor), the list size, and the list tail (the last node in the list). Public member functions might include InsertAfter (insert a new node after the given node), PushBack (insert a new node after the last node), PushFront (insert a new node at the front of the list, just after the head), DeleteNode (deletes the node from the list), etc.

Exploring further:

- More on Linked Lists from cplusplus.com

CHALLENGE ACTIVITY 8.5.1: Enter the output of the program using Linked List.

539740.3879454.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

class PlaylistSong {
   public:
      PlaylistSong(string value = "noName", PlaylistSong* nextLoc = nullptr);
      void InsertAfter(PlaylistSong* nodePtr);
      PlaylistSong* GetNext();
      void PrintNodeData();
   private:
      string name;
      PlaylistSong* nextPlaylistSongPtr;
};

PlaylistSong::PlaylistSong(string name, PlaylistSong* nextLoc) {
   this->name = name;
   this->nextPlaylistSongPtr = nextLoc;
}

void PlaylistSong::InsertAfter(PlaylistSong* nodeLoc) {
   PlaylistSong* tmpNext = nullptr;

   tmpNext = this->nextPlaylistSongPtr;
   this->nextPlaylistSongPtr = nodeLoc;
   nodeLoc->nextPlaylistSongPtr = tmpNext;
}

PlaylistSong* PlaylistSong::GetNext() {
   return this->nextPlaylistSongPtr;
}

void PlaylistSong::PrintNodeData() {
   cout << this->name << endl;
}

int main() {
   PlaylistSong* headObj = nullptr;
   PlaylistSong* firstSong = nullptr;
   PlaylistSong* secondSong = nullptr;
   PlaylistSong* thirdSong = nullptr;
   PlaylistSong* currObj = nullptr;

   headObj = new PlaylistSong("head");

   firstSong = new PlaylistSong("Egmont");
   headObj->InsertAfter(firstSong);

   secondSong = new PlaylistSong("Cavatina");
   firstSong->InsertAfter(secondSong);

   thirdSong = new PlaylistSong("Adagio");
   secondSong->InsertAfter(thirdSong);

   currObj = headObj;

   while (currObj != nullptr) {
      currObj->PrintNodeData();
      currObj = currObj->GetNext();
   }
   return 0;
}
```



```
head
Egmont
Cavatina
Adagio
```

1 2 3

Check Next

CHALLENGE ACTIVITY 8.5.2: A first linked list.

539740.3879454.qx3zqy7

Start

Two integers, chicks1 and chicks2, are read from input as the number of chicks of two chickens. headObj has the default val -1. Create a new node firstChicken with integer chicks1 and insert firstChicken after headObj. Then, create a second node secondChicken with integer chicks2 and insert secondChicken after firstChicken.

Ex: If the input is `25 12`, then the output is:

```
-1
25
12
```

```
#include <iostream>
using namespace std;

class ChickenNode {
   public:
      ChickenNode(int chicksInit = 0, ChickenNode* nextLoc = nullptr);
      void InsertAfter(ChickenNode* nodeLoc);
      ChickenNode* GetNext();
      void PrintNodeData();
   private:
      int chicksVal;
      ChickenNode* nextNodePtr;
};

ChickenNode::ChickenNode(int chicksInit, ChickenNode* nextLoc) {
```


1 2 3

Check Next level

8.6 Memory regions: Heap/Stack

A program's memory usage typically includes four different regions:

- ***Code*** — The region where the program instructions are stored.
- ***Static memory*** — The region where global variables (variables declared outside any function) as well as static local variables (variables declared inside functions starting with the keyword "static") are allocated. Static variables are allocated once and stay in the same memory location for the duration of a program's execution.
- ***The stack*** — The region where a function's local variables are allocated during a function call. A function call adds local variables to the stack, and a return removes them, like adding and removing dishes from a pile; hence the term "stack." Because this memory is automatically allocated and deallocated, it is also called ***automatic memory***.
- ***The heap*** — The region where the "new" operator allocates memory, and where the "delete" operator deallocates memory. The region is also called ***free store***.

PARTICIPATION ACTIVITY 8.6.1: Use of the four memory regions.

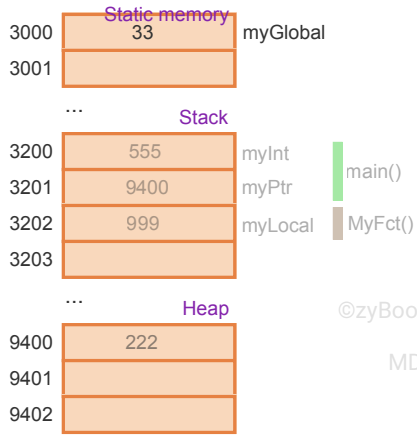

```
#include <iostream>
using namespace std;

// Program is stored in code memory

int myGlobal = 33;    // In static memory

void MyFct() {
```

Code memory

1	Add R1, #1, R2
2	Sub R3, #1, R4
3	Add R1, R3, R5
4	Jmp 40

...

```
    int myLocal;       // On stack
    myLocal = 999;
    cout << " " << myLocal;
}

int main() {
    int myInt;             // On stack
    int* myPtr = nullptr; // On stack
    myInt = 555;

    myPtr = new int;       // In heap
    *myPtr = 222;
    cout << *myPtr << " " << myInt;
    delete myPtr; // Deallocated from heap

    MyFct(); // Stack grows, then shrinks

    return 0;
}
```



Animation content:

Static figure: A code block and a column of memory locations.
Begin Cpp code:
#include <iostream>
using namespace std;

// Program is stored in code memory

int myGlobal = 33; // In static memory

void MyFct() {
  int myLocal;   // On stack
  myLocal = 999;
  cout << " " << myLocal;
}

int main() {
  int myInt;   // On stack
  int* myPtr = nullptr; // On stack
  myInt = 555;

  myPtr = new int;   // In heap
  *myPtr = 222;
  cout << *myPtr << " " << myInt;
  delete myPtr; // Deallocated from heap

  MyFct(); // Stack grows, then shrinks

  return 0;
}
End Cpp.

The program in the static figure is executed and the steps and substeps of the program are described in the following five tables. The program uses four different regions of memory along with 13 distinct memory locations. All 13 memory locations are initially empty at addresses 1 to 4, 3000 to 3001, 3200 to 3203, and 9400 to 9402. The code memory is associated with memory addresses 1 to 4, static memory is 3000 to 3001, the stack is 3200 to 3203, and the heap is 9400 to 9402.

Step 1: The code regions store program instructions. myGlobal is a global variable and is stored in the

static memory region. Code and static regions last for the entire program execution.

Code executed	Memory address and label	Value stored	Region of memory	Additional information
	1	Add R1, #1, R2	Code memory	Program instructions are added to the code memory region.
	2	Sub R3, #1, R4	Code memory	
	3	Add R1, R3, R5	Code memory	
	4	Jmp 40	Code memory	
int myGlobal = 33;	3000: myGlobal	33	Static memory	myGlobal is added to the static memory region.

Step 2: Function calls push local variables on the program stack. When main() is called, the variables myInt and myPtr are added on the stack.

Code executed	Memory address and label	Value stored	Region of memory	Additional information

int main() {
  int myInt;
  int* myPtr = nullptr;
  myInt = 555;
	3200: myInt	555	Stack	memory address 3200 myInt and memory address 3201 myPtr are grouped with the label main()

	3201: myPtr	nullPtr	Stack	

Step 3: new allocates memory on the heap for an int and returns the address of the allocated memory, which is assigned to myPtr. delete deallocates memory from the heap.

Code executed	Memory address and label	Value stored	Region of memory	Additional information
myPtr = new int;	3201: myPtr	9400	Stack	Memory address 9400 in the Heap now has the label (int)
*myPtr = 222;	9400: (int)	222	Heap	
cout << *myPtr << " " << myInt;				
delete myPtr;	9400		Heap	The (int) label disappears and the value 222 in memory location 9400 fades.

Step 4: Calling MyFct() grows the stack, pushing the function's local variables on the stack. Those

local variables are removed from the stack when the function returns.

Code executed	Memory address and label	Value stored	Region of memory	Additional information
MyFct();				The function MyFct() is called.

void MyFct() {
int myLocal;
myLocal = 999;
	3202: myLocal	999	Stack	memory address 3202 myLocal has the label MyFct()

cout << " " << myLocal; }	3202		Stack	After MyFct() ends, the value 999 and labels myLabel and MyFct() are faded in the Stack.
MyFct();				The program returns back to the function call.

Step 5: When main() completes, main's local variables are removed from the stack.

Code executed	Memory address and label	Value stored	Region of memory	Additional information
return 0;	3200 and 3201		Stack	After main() ends, the value and label 555 myInt and 9400 myPtr and the label main() are all faded in the Stack.

Animation captions:

1. The code regions store program instructions. myGlobal is a global variable and is stored in the static memory region. Code and static regions last for the entire program execution.
2. Function calls push local variables on the program stack. When main() is called, the variables myInt and myPtr are added on the stack.
3. new allocates memory on the heap for an int and returns the address of the allocated memory, which is assigned to myPtr. delete deallocates memory from the heap.
4. Calling MyFct() grows the stack, pushing the function's local variables on the stack. Those local variables are removed from the stack when the function returns.
5. When main() completes, main's local variables are removed from the stack.

PARTICIPATION ACTIVITY 8.6.2: Stack and heap definitions.

If unable to drag and drop, refresh the page.

Automatic memory | **Code** | **Free store** | **Static memory** | **The heap** | **The stack**

	A function's local variables are allocated in this region while a function is called.

	The memory allocation and deallocation operators affect this region.
	Global and static local variables are allocated in this region once for the duration of the program.
	Another name for "The heap" because the programmer has explicit control of this memory.
	Instructions are stored in this region.
	Another name for "The stack" because the programmer does not explicitly control this memory.

Reset

8.7 Destructors

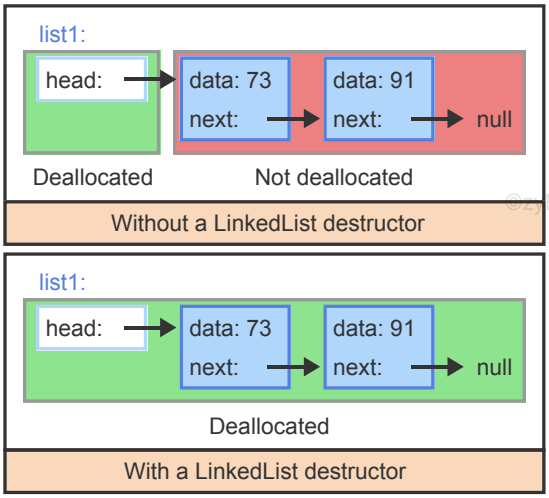
Overview

A ***destructor*** is a special class member function that is called automatically when a variable of that class type is destroyed. C++ class objects commonly use dynamically allocated data that is deallocated by the class's destructor.

Ex: A linked list class dynamically allocates nodes when adding items to the list. Without a destructor, the link list's nodes are not deallocated. The linked list class destructor should be implemented to deallocate each node in the list.

PARTICIPATION ACTIVITY | 8.7.1: LinkedList nodes are not deallocated without a LinkedList class destructor.

```
class LinkedListNode {
public:
   ...
   int data;
   LinkedListNode* next;
};

class LinkedList {
public:
   ...
   LinkedListNode* head;
};

int main() {
   LinkedList* list1;
   list1 = new LinkedList();
   ... // Add items to list1
   delete list1;
}
```



Animation content:

Static figure: A code block and two rows of linked lists.
Begin Cpp code:

```
class LinkedListNode {
public:
  ...
  int data;
  LinkedListNode* next;
};

class LinkedList {
public:
  ...
  LinkedListNode* head;
};

int main() {
  LinkedList* list1;
  list1 = new LinkedList();
  ... // Add items to list1
  delete list1;
}
```

End Cpp code.
Step 1: The LinkedList class has a pointer to the list's head, initially set to null by the LinkedList class constructor.
The code LinkedListNode* head; is highlighted, followed by the two lines of code:
LinkedList* list1;
list1 = new LinkedList();
The first linked list appears with the label list1. The head is enclosed in a rectangle with an arrow heading to the word null.
Step 2: After adding 2 items, 3 dynamically allocated objects exist: the list itself and 2 nodes.
The following code is highlighted: ... // Add items to list1. The head of the linked list now points to the first node with data: 73 and next pointing to the second node. The second node has data: 91 and next pointing to null.
Step 3: Without a destructor, deleting list1 only deallocates the list1 object, but not the 2 nodes.
The code delete list1; is highlighted. The head is enclosed in a green rectangle labeled Deallocated, and the two nodes are enclosed in a red rectangle labeled Not deallocated. The entire linked list is labeled: Without a LInkedList destructor.
Step 4: With a properly implemented destructor, the LinkedList class will free the list's nodes.
The code delete list1; is still highlighted. The second linked list appears. It is identical to the first, but this linked list is entirely enclosed in a green rectangle labeled Deallocated. The linked list is labeled: With a Linkedlist destructor.

Animation captions:

1. The LinkedList class has a pointer to the list's head, initially set to null by the LinkedList class constructor.
2. After adding 2 items, 3 dynamically allocated objects exist: the list itself and 2 nodes.
3. Without a destructor, deleting list1 only deallocates the list1 object, but not the 2 nodes.
4. With a properly implemented destructor, the LinkedList class will free the list's nodes.

PARTICIPATION ACTIVITY 8.7.2: LinkedList class destructor.

1) Using the delete operator to deallocate a LinkedList object automatically frees all nodes allocated by that object.
   - True
   - False

2) A destructor for the LinkedList class would be implemented as a LinkedList class member function.
   - True
   - False

3) If list1 were declared without dynamic allocation, as shown below, no destructor would be needed.

```
LinkedList list1;
```

   - True
   - False

Implementing the LinkedList class destructor

The syntax for a class's destructor function is similar to a class's constructor function, but with a "~" (called a "tilde" character) prepended to the function name. A destructor has no parameters and no return value. So the LinkedListNode and LinkedList class destructors are declared as `~LinkedListNode();` and `~LinkedList();`, respectively.

The LinkedList class destructor is implemented to free each node in the list. The LinkedListNode destructor is not required, but is implemented below to display a message when a node's destructor is called. Using delete to free a dynamically allocated LinkedListNode or LinkedList will call the object's destructor.

Figure 8.7.1: LinkedListNode and LinkedList classes.

```
#include <iostream>
using namespace std;

class LinkedListNode {
public:
   LinkedListNode(int dataValue) {
      cout << "In LinkedListNode constructor (" << dataValue << ")" <<
endl;
      data = dataValue;
   }

   ~LinkedListNode() {
      cout << "In LinkedListNode destructor (";
      cout << data << ")" << endl;
   }

   int data;
   LinkedListNode* next;
};

class LinkedList {
public:
   LinkedList();
   ~LinkedList();
   void Prepend(int dataValue);

   LinkedListNode* head;
};

LinkedList::LinkedList() {
   cout << "In LinkedList constructor" << endl;
   head = nullptr;
}

LinkedList::~LinkedList() {
   cout << "In LinkedList destructor" << endl;

   // The destructor deletes each node in the linked list
   while (head) {
      LinkedListNode* next = head->next;
      delete head;
      head = next;
   }
}

void LinkedList::Prepend(int dataValue) {
   LinkedListNode* newNode = new LinkedListNode(dataValue);
   newNode->next = head;
   head = newNode;
}
```


PARTICIPATION ACTIVITY 8.7.3: The LinkedList class destructor, called when the list is deleted, frees all nodes.


```
#include <iostream>
using namespace std;

// ... LinkedListNode class omitted ...
// ... LinkedList class omitted, except for destructor, below ...

LinkedList::~LinkedList() {
   cout << "In LinkedList destructor" << endl;

   // The destructor deletes each node in the linked list
```

Console:

```
In LinkedList constructor
In LinkedListNode constructor (10)
In LinkedListNode constructor (20)
In LinkedListNode constructor (30)
In LinkedListNode constructor (40)
In LinkedListNode constructor (50)
In LinkedList destructor
In LinkedListNode destructor (50)
In LinkedListNode destructor (40)
```

```
    while (head) {
       LinkedListNode* next = head->next;
       delete head;
       head = next;
    }
 }

 int main() {
    // Create a linked list
    LinkedList* list = new LinkedList;
    for (int i = 1; i <= 5; ++i)
       list->Prepend(i * 10);

    // Free the linked list.
    // The LinkedList class destructor frees each node.
    delete list;

    return 0;
 }
```

```
In LinkedListNode destructor (30)
In LinkedListNode destructor (20)
In LinkedListNode destructor (10)
```


Animation content:

Static figure: A code block and a console.
Begin Cpp code:

```
#include <iostream>
using namespace std;

// ... LinkedListNode class omitted ...
// ... LinkedList class omitted, except for destructor, below ...

LinkedList::~LinkedList() {
  cout << "In LinkedList destructor" << endl;

  // The destructor deletes each node in the linked list
  while (head) {
    LinkedListNode* next = head->next;
    delete head;
    head = next;
  }
}

int main() {
  // Create a linked list
  LinkedList* list = new LinkedList;
  for (int i = 1; i ≤ 5; ++i)
    list->Prepend(i * 10);

  // Free the linked list.
  // The LinkedList class destructor frees each node.
  delete list;

  return 0;
```

End Cpp code.

The program in the static figure is executed and the steps and substeps of the program are described in the following four tables.

Step 1: A linked list is created and 5 dynamically allocated nodes are prepended.

Code executed	Output on Console

LinkedList* list = new LinkedList;	In LinkedIst constructor
for (int i = 1; i ≤ 5; ++i) list->Prepend(i * 10);	In LinkedIstNode constructor(10) In LinkedIstNode constructor(20) In LinkedIstNode constructor(30) In LinkedIstNode constructor(40) In LinkedIstNode constructor(50)

Step 2: Deleting the list calls the LinkedList class destructor.

Code executed	Output on Console
delete list;	
LinkedList::~LinkedList() { cout << "In LinkedList destructor" << endl;	In LinkedList destructor

Step 3: The destructor deletes each node in the list.

Code executed	Output on Console

while (head) {
LinkedListNode* next = head->next;
delete head;
head = next;
}
	In LinkedIstNode destructor(50)
	In LinkedIstNode destructor(40)
	In LinkedIstNode destructor(30)
	In LinkedIstNode destructor(20)
	In LinkedIstNode destructor(10)

Step 4: After calling ~LinkedList(), the delete operator frees memory for the linked list's head pointer. All memory for the linked list has been freed.

Code executed	Output on Console
delete list;	
return 0;	

Animation captions:

1. A linked list is created and 5 dynamically allocated nodes are prepended.
2. Deleting the list calls the LinkedList class destructor.
3. The destructor deletes each node in the list.
4. After calling ~LinkedList(), the delete operator frees memory for the linked list's head pointer. All memory for the linked list has been freed.

PARTICIPATION ACTIVITY 8.7.4: LinkedList class destructor.

1) After ~LinkedList() is called, the list's head pointer points to _____.

- null
- the first node, which is now freed
- the last node, which is now freed

2) When ~LinkedList() is called, ~LinkedListNode() gets called for each node in the list.

- True
- False

3) If the LinkedList class were renamed to just List, the destructor function must be redeclared as _____.

- `void ~List();`
- `~List();`
- `List();`

When a destructor is called

Using the delete operator on an object allocated with the new operator calls the destructor, as shown in the previous example. For an object that is not declared by reference or by pointer, the object's destructor is called automatically when the object goes out of scope.

PARTICIPATION ACTIVITY 8.7.5: Destructors are called automatically only for non-reference/pointer variables.

```
int main() {
   LinkedList list1;
   list1.Prepend(1);

   cout << "Exiting main" << endl;
   return 0;
}
```

Console:

```
In LinkedList constructor
In LinkedListNode constructor (1)
Exiting main
In LinkedList destructor
In LinkedListNode destructor (1)
```

list1's destructor is called

```
int main() {
   LinkedList* list2 = new LinkedList();
   list2->Prepend(2);

   cout << "Exiting main" << endl;
   return 0;
}
```

Console:

```
In LinkedList constructor
In LinkedListNode constructor (2)
Exiting main
```

list2's destructor is not called

```
int main() {
   LinkedList& list3 = *(new LinkedList());
   list3.Prepend(3);

   cout << "Exiting main" << endl;
   return 0;
}
```

Console:

```
In LinkedList constructor
In LinkedListNode constructor (3)
Exiting main
```

list3's destructor is not called

Animation content:

Static figure: Three rows with one code block and one console in each.
Begin Cpp code:
int main() {
LinkedList list1;

```
  list1.Prepend(1);

  cout << "Exiting main" << endl;
  return 0;
}
End Cpp code.

Begin Cpp code:
int main() {
  LinkedList* list2 = new LinkedList();
  list2->Prepend(2);

  cout << "Exiting main" << endl;
  return 0;
}
End Cpp code.

Begin Cpp code:
int main() {
  LinkedList& list3 = *(new LinkedList());
 list3.Prepend(3);

  cout << "Exiting main" << endl;
  return 0;
}
End Cpp code.
```


The program in the static figure is executed and the steps and substeps of the program are described in the following four tables.

Step 1: list1 is declared as a local variable and is not a pointer or reference.

Code executed	Output on Console	Additional Information
LinkedList list1;	In LinkedList constructor In LinkedListNode constructor (1)	From first code block and console

Step 2: list1 goes out of scope at the end of main(). So list1's destructor is called automatically.

Code executed	Output on Console	Additional Information
cout << "Exiting main" << endl;	Exiting main	
}	In LinkedList destructor In LinkedListNode destructor (1)	The following label appears in green under the first console: list1's destructor is called

Step 3: list2 is declared as a pointer and the destructor is not automatically called at the end of main().

Code executed	Output on Console	Additional Information
int main() { LinkedList* list2 = new LinkedList();	In LinkedList constructor In LinkedListNode	The following label appears in red under the second console: list2's destructor is not called

list2->Prepend(2);
 	
cout << "Exiting main" << endl;
return 0;
}
	constructor (2)
	Exiting main	

Step 4: list3 is declared as a reference and the destructor is not automatically called at the end of main().

Code executed	Output on Console	Additional Information

int main() {
LinkedList& list3 = *(new LinkedList());
list3.Prepend(3);
 	
cout << "Exiting main" << endl;
return 0;
}
	In LinkedList constructor
	In LinkedListNode constructor (3)
	Exiting main
		The following label appears in red under the third console: list3's destructor is not called

Animation captions:

1. list1 is declared as a local variable and is not a pointer or reference.
2. list1 goes out of scope at the end of main(). So list1's destructor is called automatically.
3. list2 is declared as a pointer and the destructor is not automatically called at the end of main().
4. list3 is declared as a reference and the destructor is not automatically called at the end of main().

PARTICIPATION ACTIVITY 8.7.6: When a destructor is called.

1) Both the constructor and destructor are called by the following code.

```
delete (new LinkedList());
```

- True
- False

2) listToDisplay's destructor is called at the end of the DisplayList function.

```
void DisplayList(LinkedList listToDisplay) {
   LinkedListNode* node = listToDisplay.head;
   while(node) {
      cout << node->data << " ";
      node = node->next;
   }
}
```


- True
- False

3) listToDisplay's destructor is called at the end of the DisplayList function.

```
void DisplayList(LinkedList&
listToDisplay) {
   LinkedListNode* node =
listToDisplay.head;
   while(node) {
      cout << node->data << " ";
      node = node->next;
   }
}
```

- True
- False

CHALLENGE ACTIVITY 8.7.1: Enter the output of the destructors.

539740.3879454.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

class IntNode {
   public:
      IntNode(int value) {
         numVal = value;
      }

      ~IntNode() {
         cout << numVal << endl;
      }

      int numVal;
};

int main() {
   IntNode* node1 = new IntNode(1);
   IntNode* node2 = new IntNode(3);
   IntNode* node3 = new IntNode(6);
   IntNode* node4 = new IntNode(8);

   delete node3;
   delete node2;
   delete node1;
   delete node4;

   return 0;
}
```

6
3
1
8

1 2

Check Next

CHALLENGE ACTIVITY 8.7.2: Destructors.

539740.3879454.qx3zqy7

Start

Complete the PigNode class destructor. The destructor prints "PigNode with ", followed by the value of piglets, then " piglets is deallocated.". End with a newline.

Ex: If the input is 7, then the output is:

```
PigNode with 7 piglets is deallocated.
```

```
#include <iostream>
using namespace std;

class PigNode {
   public:
      PigNode(int pigletsValue) {
         piglets = pigletsValue;
      }

      /* Your code goes here */

      int piglets;
      PigNode* next;
};
```


1 2

Check Next level

Exploring further:

- More on Destructors from msdn.microsoft.com
- Order of Destruction from msdn.microsoft.com

8.8 Memory leaks

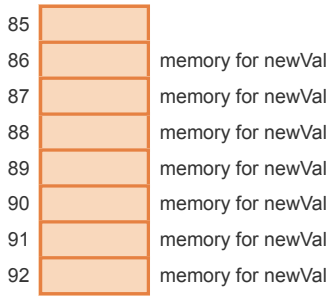
Memory leak

A ***memory leak*** occurs when a program that allocates memory loses the ability to access the allocated memory, typically due to failure to properly destroy/free dynamically allocated memory. A program's leaking memory becomes unusable, much like a water pipe might have water leaking out and becoming unusable. A memory leak may cause a program to occupy more and more memory as the program runs, which slows program runtime. Even worse, a memory leak can cause the program to fail if memory becomes completely full and the program is unable to allocate additional memory.

A common error is failing to free allocated memory that is no longer used, resulting in a memory leak. Many programs that are commonly left running for long periods, like web browsers, suffer from known memory leaks — a web search for "<your-favorite-browser> memory leak" will likely result in numerous hits.

PARTICIPATION ACTIVITY 8.8.1: Memory leak can use up all available memory.

```
while (expression) {
   // Allocate memory for newVal
   // Do something with newVal
   // Not deallocating newVal
   // results in memory leak
}
```



Animation content:

Static figure: A code block and a column of memory locations.
Begin Cpp code:
while (expression) {
  // Allocate memory for newVal
  // Do something with newVal
  // Not deallocating newVal
  // results in memory leak
}
End Cpp code.
The 13 memory locations, addresses 75 to 87, are initially empty.
Step 1: Memory is allocated for newVal each loop iteration, but the loop does not deallocate memory once done using newVal, resulting in a memory leak.
The code while (expression) is highlighted, followed by the comment // Allocate memory for newVal. The text memory for newVal appears as the label of memory location 86. The comment // Do something with newVal is highlighted, followed by the two comments:
// Not deallocating newVal
// results in memory leak
Step 2: Each loop iteration allocates more memory, eventually using up all available memory and causing the program to fail.
The comment // Allocate memory for newVal is highlighted. The text memory for newVal appears as the label of memory location 87. The comment // Do something with newVal is highlighted, followed by the two comments:
// Not deallocating newVal
// results in memory leak
The four comments in the while loop are highlighted as the text memory for newVal labels memory locations 88 to 92, one at a time.

Animation captions:

1. Memory is allocated for newVal each loop iteration, but the loop does not deallocate memory once done using newVal, resulting in a memory leak.
2. Each loop iteration allocates more memory, eventually using up all available memory and causing the program to fail.

Garbage collection

*Some programming languages, such as Java, use a mechanism called **garbage collection** wherein a program's executable includes automatic behavior that at various intervals finds all unreachable allocated memory locations (e.g., by comparing all reachable memory with all previously-allocated memory), and automatically frees such unreachable memory. Some non-standard C++ implementations also include garbage collection. Garbage collection can reduce the impact of memory leaks at the expense of runtime overhead. Computer scientists debate whether new programmers should learn to explicitly free memory versus letting garbage collection do the work.*

PARTICIPATION ACTIVITY 8.8.2: Memory leaks.

If unable to drag and drop, refresh the page.

Memory leak **Garbage collection** **Unusable memory**

	Memory locations that have been dynamically allocated but can no longer be used by a program.
	Occurs when a program allocates memory but loses the ability to access the allocated memory.
	Automatic process of finding and freeing unreachable allocated memory locations.

Reset

Memory not freed in a destructor

Destructors are needed when destroying an object involves more work than simply freeing the object's memory. Such a need commonly arises when an object's data member, referred to as a sub-object, has allocated additional memory. Freeing the object's memory without also freeing the sub-object's memory results in a problem where the sub-object's memory is still allocated, but inaccessible, and thus can't be used again by the program.

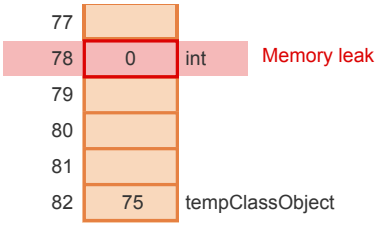
The program in the animation below is very simple to focus on how memory leaks occur with sub-objects. The class's sub-object is just an integer pointer but typically would be a pointer to a more complex type. Likewise, the object is created and then immediately destroyed, but typically something would have been done with the object.

PARTICIPATION ACTIVITY 8.8.3: Lack of destructor yields memory leak.

```
#include <iostream>
using namespace std;
```

75 78 subObject MyClass
76

```
class MyClass {
   public:
      MyClass();
   private:
      int* subObject;
};

MyClass::MyClass() {
   subObject = new int;    // Allocate sub-object
   *subObject = 0;
}


int main() {
   MyClass* tempClassObject;

   tempClassObject = new MyClass;
   delete tempClassObject; // ERROR: Memory leak
   // Freed obj's mem, but not subobj's

   // Rest of program ...

   return 0;
}
```



Animation content:

Static figure: A code block and a column of memory locations.
Begin Cpp code:
#include <iostream>
using namespace std;

class MyClass {
  public:
    MyClass();
  private:
    int* subObject;
};

MyClass::MyClass() {
  subObject = new int;   // Allocate sub-object
  *subObject = 0;
}

int main() {
  MyClass* tempClassObject;

  tempClassObject = new MyClass;
  delete tempClassObject; // ERROR: Memory leak
  // Freed obj's mem, but not subobj's

  // Rest of program ...

  return 0;
}
End Cpp code.

The program in the static figure is executed and the steps and substeps of the program are described in the following three tables. The 8 memory locations, addresses 75 to 82, are initially empty.

Step 1: tempClassObject is a pointer to object of type MyClass. new allocates memory for the object.

Code executed	Memory address and label	Value stored	Additional information
MyClass* tempClassObject;	82: tempClassObject		
tempClassObject = new MyClass; int* subObject;	75: subObject		Memory address 75 subObject has the label MyClass.

Step 2: The constructor for the MyClass object is called. The constructor allocates memory for an int using the pointer subObject.

Code executed	Memory address and label	Value stored	Additional information
subObject = new int;	75: subObject	78	Memory location 78 has the label int.
*subObject = 0;	78: int	0	
tempClassObject = new MyClass;	82: tempClassObject	75	

Step 3: Deleting tempClassObject frees the memory for the tempClassObject, but not subObject. A memory leak results because memory location 78 is still allocated, but nothing points to the memory allocation.

Code executed	Memory address and label	Value stored	Additional information
delete tempClassObject;	75		The subObject label, MyClass label, and the value 78 in memory location 75 all fade. Memory location 78 with value 0 and label int is highlighted red and now has the label Memory leak.

Animation captions:

1. tempClassObject is a pointer to object of type MyClass. new allocates memory for the object.
2. The constructor for the MyClass object is called. The constructor allocates memory for an int using the pointer subObject.
3. Deleting tempClassObject frees the memory for the tempClassObject, but not subObject. A memory leak results because memory location 78 is still allocated, but nothing points to the memory allocation.

PARTICIPATION ACTIVITY 8.8.4: Memory not freed in a destructor.

1) In the above animation, which object's memory is not freed?

- `MyClass`
- `tempClassObject`
- `subObject`

2) Does a memory leak remain when the above program terminates?

- Yes
- No

3) What line must exist in `MyClass`'s destructor to free all memory allocated by a `MyClass` object?

- `delete subObject;`
- `delete tempClassObject;`
- `delete MyClass;`

PARTICIPATION ACTIVITY 8.8.5: Which results in a memory leak?

Which scenario results in a memory leak?

1)
```
int main() {
   MyClass* ptrOne = new
MyClass;
   MyClass* ptrTwo = new
MyClass;

   ptrOne = ptrTwo;
   return 0;
}
```

- Memory leak
- No memory leak

2)
```
int main() {
   MyClass* ptrOne = new
MyClass;
   MyClass* ptrTwo = new
MyClass;
   MyClass* ptrThree;

   ptrThree = ptrOne;
   ptrOne = ptrTwo;
   return 0;
}
```

- Memory leak
- No memory leak

3)
```
class MyClass {
   public:
      MyClass() {
         subObject = new int;
         *subObject = 0;
      }

      ~MyClass() {
         delete subObject;
      }

   private:
      int* subObject;
};

int main() {
   MyClass* ptrOne = new
MyClass;
   MyClass* ptrTwo = new
MyClass;
   ...

   delete ptrOne;
   ptrOne = ptrTwo;
   return 0;
}
```

- Memory leak
- No memory leak

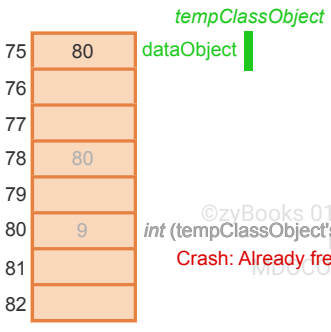
8.9 Copy constructors

Copying an object without a copy constructor

The animation below shows a typical problem that arises when an object is passed by value to a function and no copy constructor exists for the object.

PARTICIPATION ACTIVITY 8.9.1: Copying an object without a copy constructor.

```
class MyClass {
  public:
     MyClass() {
        cout << "Constructor called." << endl;
        dataObject = new int; // Allocate data object
        *dataObject = 0;
     }
     ~MyClass() {
        cout << "Destructor called." << endl;
        delete dataObject;
     }
     void SetDataObject(const int i) { *dataObject = i; }
     int GetDataObject() const { return *dataObject; }

  private:
     int* dataObject;
};

void SomeFunction(MyClass localObject) {
  // Do something with localObject
}

int main() {
  MyClass tempClassObject; // Create object of type MyClass
```




```
Constructor called.
Before: 9
Destructor called.
After: 0
Destructor called.
```

```
   // Set and print data member value
   tempClassObject.SetDataObject(9);
   cout << "Before: " << tempClassObject.GetDataObject() << endl;

   // Calls SomeFunction(), tempClassObject is passed by value
   SomeFunction(tempClassObject);

   cout << "After: " << tempClassObject.GetDataObject() << endl;

   return 0;
}
```


Animation content:

Static figure:
Begin Cpp code:

```
class MyClass {
 public:
   MyClass() {
     cout << "Constructor called." << endl;
     dataObject = new int; // Allocate data object
     *dataObject = 0;
   }
   ~MyClass() {
     cout << "Destructor called." << endl;
     delete dataObject;
   }
   void SetDataObject(const int i) { *dataObject = i; }
   int GetDataObject() const { return *dataObject; }

 private:
   int* dataObject;
};

void SomeFunction(MyClass localObject) {
 // Do something with localObject
}

int main() {
 MyClass tempClassObject; // Create object of type MyClass

 // Set and print data member value
 tempClassObject.SetDataObject(9);
 cout << "Before: " << tempClassObject.GetDataObject() << endl;

 // Calls SomeFunction(), tempClassObject is passed by value
 SomeFunction(tempClassObject);

 cout << "After: " << tempClassObject.GetDataObject() << endl;

 return 0;
}
```

End Cpp code.

Memory addresses 75 to 82 are shown. The variable, tempClassObject, is at memory address 75,

©zyBooks 01/31/24 17:50 1939727
Rob Daglio
MDCCOP2335Spring2024

contains the value 80, and is a dataObject. The value, 80, is at memory address 78. The value, 9, is at memory address 80, is labeled, tempClassObject's, and is an int. Memory address 80 is labeled Crash: Already freed. The output consosle has the lines:

Constructor called.
Before: 9
Destructor called.
After: 0
Destructor called.

Step 1: The constructor creates object tempClassObject and sets the object's dataObject (a pointer) to the value 9. The value 9 is printed.

The line of code, MyClass tempClassObject;, is highlighted. Memory address 75 contains the variable, tempClassObject, and is a dataObject. The line of code, MyClass() {, is highlighted. The line of code, cout << "Constructor called." << endl;, is highlighted. "Constructor called." is output to the console. The line of code, dataObject = new int; // Allocate data object, is highlighted, and memory addres 80 is labeled, tempClassObject's, and is an int. The memory address 75 obtains the value 80. Memory address 80 obtains the value 0. The line of code, tempClassObject.SetDataObject(9);, is highlighted. The value at memory address 80 is updated from 0 to 9. The line of code, cout << "Before: " << tempClassObject.GetDataObject() << endl;, is highlighted. "Before: 9" is output to the console.

Step 2: SomeFunction() is called and tempClassObject is passed by value, creating a local copy of the object with the same dataObject.

The line of code, SomeFunction(tempClassObject);, is highlighted. The line of code, void SomeFunction(MyClass localObject) {, is highlighted. Memory address 78 obtains the variable, localObject, is a dataObject, nd contains the value 80.

Step 3: When SomeFunction() returns, localObject is destroyed and the MyClass destructor frees the dataObject's memory. tempClassObject's dataObject value is changed and now 0 is printed.

The line of code, }, is highlighted. The line of code, ~MyClass() {, is highlighted. The line of code, cout << "Destructor called." << endl;, is highlighted. Destructor called. is output to the console. The line of code, delete dataObject;, is highlighted. The value, 80, is faded out at memory address 78, the dataObject, localObject, is deleted, the value 9 at memory address 80 is faded, and memory adderss 80 is labeled Freed. The line of code, cout << "After: " << tempClassObject.GetDataObject() << endl;, is highlighted and "After: 0" is output to the console.

Step 4: When main() returns, the MyClass destructor is called again, attempting to free the dataObject's memory again, causing the program to crash.

The line of code, return 0;, is highlighted. The line of code, ~MyClass() {, is highlighted. The line of code, cout << "Destructor called." << endl;, is highlighted. Destructor called. is output to the console. The line of code, delete dataObject;, is highlighted. Memory address 80 is updated from being labeled Freed to Crash: Already freed.

Animation captions:

1. The constructor creates object tempClassObject and sets the object's dataObject (a pointer) to the value 9. The value 9 is printed.
2. SomeFunction() is called and tempClassObject is passed by value, creating a local copy of the object with the same dataObject.

3. When SomeFunction() returns, localObject is destroyed and the MyClass destructor frees the dataObject's memory. tempClassObject's dataObject value is changed and now 0 is printed.
4. When main() returns, the MyClass destructor is called again, attempting to free the dataObject's memory again, causing the program to crash.

PARTICIPATION ACTIVITY 8.9.2: Copying an object without a copy constructor.

1) If an object with an int sub-object is passed by value to a function, the program will complete execution with no errors.
   - True
   - False

2) If an object with an int* sub-object is passed by value to a function, the program will complete execution with no errors.
   - True
   - False

3) If an object with an int* sub-object is passed by value to a function, the program will call the class constructor to create a local copy of the sub-object.
   - True
   - False

Copy constructor

The solution is to create a ***copy constructor***, a constructor that is automatically called when an object of the class type is passed by value to a function and when an object is initialized by copying another object during declaration. Ex: `MyClass classObj2 = classObj1;` or `obj2Ptr = new MyClass(classObj1);`. The copy constructor makes a new copy of all data members (including pointers), known as a ***deep copy***.

If the programmer doesn't define a copy constructor, then the compiler implicitly defines a constructor with statements that perform a memberwise copy, which simply copies each member using assignment: `newObj.member1 = origObj.member1`, `newObj.member2 = origObj.member2`, etc. Creating a copy of an object by copying only the data members' values creates a ***shallow copy*** of the object. A shallow copy is fine for many classes, but typically a deep copy is desired for objects that have data members pointing to dynamically allocated memory.

The copy constructor can be called with a single pass-by-reference argument of the class type, representing an original object to be copied to the newly-created object. A programmer may define a copy constructor, typically having the form:

```
MyClass(const MyClass& origObject);
```


Construct 8.9.1: Copy constructor.

```
class MyClass {
   public:
      ...
      MyClass(const MyClass&
origObject);
      ...
};
```


The program below adds a copy constructor to the earlier example, which makes a deep copy of the data member dataObject within the MyClass object. The copy constructor is automatically called during the call to SomeFunction(). Destruction of the local object upon return from SomeFunction() frees the newly created dataObject for the local object, leaving the original tempClassObject's dataObject untouched. Printing after the function call correctly prints 9, and destruction of tempClassObject during the return from main() produces no error.

Figure 8.9.1: Problem solved by creating a copy constructor that does a deep copy.

```
#include <iostream>
using namespace std;

class MyClass {
public:
   MyClass();
   MyClass(const MyClass& origObject); // Copy constructor
   ~MyClass();

   // Set member value dataObject
   void SetDataObject(const int setVal) {
      *dataObject = setVal;
   }

   // Return member value dataObject
   int GetDataObject() const {
      return *dataObject;
   }
private:
   int* dataObject;// Data member
};

// Default constructor
MyClass::MyClass() {
   cout << "Constructor called." << endl;
   dataObject = new int; // Allocate mem for data
   *dataObject = 0;
}

// Copy constructor
MyClass::MyClass(const MyClass& origObject) {
   cout << "Copy constructor called." << endl;
   dataObject = new int; // Allocate sub-object
   *dataObject = *(origObject.dataObject);
}

// Destructor
MyClass::~MyClass() {
   cout << "Destructor called." << endl;
   delete dataObject;
}

void SomeFunction(MyClass localObj) {
   // Do something with localObj
}

int main() {
   MyClass tempClassObject; // Create object of type MyClass

   // Set and print data member value
   tempClassObject.SetDataObject(9);
   cout << "Before: " << tempClassObject.GetDataObject() << endl;

   // Calls SomeFunction(), tempClassObject is passed by value
   SomeFunction(tempClassObject);

   // Print data member value
   cout << "After: " << tempClassObject.GetDataObject() << endl;

   return 0;
}
```

```
Constructor called.
Before: 9
Copy constructor called.
Destructor called.
After: 9
Destructor called.
```

Copy constructors in more complicated situations

The above examples use a trivially-simple class having a dataObject whose type is a pointer to an integer, to focus attention on the key issue. Real situations typically involve classes with multiple data members and with data objects whose types are pointers to class-type objects.

PARTICIPATION ACTIVITY 8.9.3: Determining which constructor will be called.

Given the following class declaration and variable declaration, determine which constructor will be called for each of the following statements.

```
class EncBlock {
public:
   EncBlock();                        // Default constructor
   EncBlock(const EncBlock& origObj); // Copy constructor
   EncBlock(int blockSize);           // Constructor with int parameter
   ~EncBlock();                       // Destructor
   ...
};

EncBlock myBlock;
```

1) `EncBlock* aBlock = new EncBlock(5);`

- `EncBlock();`
- `EncBlock(const EncBlock& origObj);`
- `EncBlock(int blockSize);`

2) `EncBlock testBlock;`

- `EncBlock();`
- `EncBlock(const EncBlock& origObj);`
- `EncBlock(int blockSize);`

3) `EncBlock* lastBlock = new EncBlock(myBlock);`

- `EncBlock();`
- `EncBlock(const EncBlock& origObj);`
- `EncBlock(int blockSize);`

4) `EncBlock vidBlock = myBlock;`

- `EncBlock();`
- `EncBlock(const EncBlock& origObj);`
- `EncBlock(int blockSize);`

Exploring further:

- More on Copy Constructors from cplusplus.com

CHALLENGE ACTIVITY 8.9.1: Enter the output of the copy constructors.

539740.3879454.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

class IntNode {
   public:
      IntNode(int value) {
         numVal = new int;
         *numVal = value;
      }
      void SetNumVal(int val) { *numVal = val; }
      int GetNumVal() { return *numVal; }
   private:
      int* numVal;
};

int main() {
   IntNode node1(1);
   IntNode node2(2);
   IntNode node3(3);

   node1 = node2;
   node2.SetNumVal(9);

   cout << node1.GetNumVal() << " " << node2.GetNumVal() << endl;

   return 0;
}
```

9 9

1 2

Check Next

CHALLENGE ACTIVITY 8.9.2: Write a copy constructor.

Write a copy constructor for CarParkingSpot that assigns origCarParkingSpot.parkingSpot to the constructed object's parkingSpot. Sample output for the given program:


```
Parking Spot: 5
```

Learn how our autograder works

539740.3879454.qx3zqy7

```
1 #include <iostream>
2 using namespace std;
3
```

```
class CarParkingSpot {
   public:
      CarParkingSpot();
      CarParkingSpot(const CarParkingSpot& origCarParkingSpot);
      void SetParkingSpot(const int spot) {
         *parkingSpot = spot;
      }
      int GetParkingSpot() const {
         return *parkingSpot;
      }
   private:
      int* parkingSpot;
```


Run

CHALLENGE ACTIVITY | 8.9.3: Copy constructors.

539740.3879454.qx3zqy7

Start

AnimalColony is a class with one int* and one double* data member pointing to the size of population and rate of growth of animal colony, respectively. An integer and a double are read from input to initialize elkColony. Use the copy constructor to cr an AnimalColony object named copyColony that is a deep copy of elkColony.

Ex: If the input is 21 0.05, then the output is:

```
Original constructor called
Constructed a new AnimalColony object
elkColony: 21 elks with 5.00% rate of growth
copyColony: 42 elks with 10.00% rate of growth
```

```
#include <iostream>
#include <iomanip>
using namespace std;

class AnimalColony {
   public:
      AnimalColony(int startingSize = 0, double startingRate = 0.0);
      AnimalColony(const AnimalColony& col);
      void SetSize(int newSize);
      void SetRate(double newRate);
      int GetSize() const;
      double GetRate() const;
      void Print() const;
   private:
      int* size;
```


1 2

Check Next level

8.10 Copy assignment operator

Default assignment operator behavior

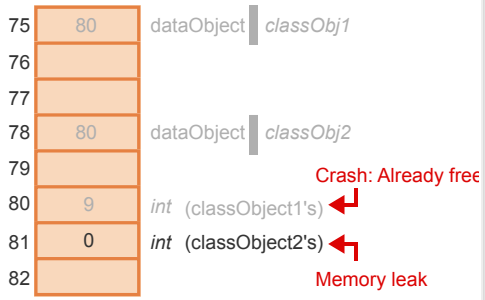
Given two MyClass objects, classObj1 and classObj2, a programmer might write `classObj2 = classObj1;` to copy classObj1 to classObj2. The default behavior of the assignment operator (=) for classes or structs is to perform memberwise assignment. Ex:


```
classObj2.memberVal1 = classObj1.memberVal1;
classObj2.memberVal2 = classObj1.memberVal2;
...
```

Such behavior may work fine for members with basic types like int or char, but typically is not the desired behavior for a pointer member. Memberwise assignment of pointers may lead to program crashes or memory leaks.

PARTICIPATION ACTIVITY 8.10.1: Basic assignment operation fails when pointer member involved.

```
#include <iostream>
using namespace std;

class MyClass {
public:
   MyClass();
   ~MyClass();
   void SetDataObject(const int i) {*dataObject = i;}
   int GetDataObject() const {return *dataObject;}

private:
   int* dataObject;
};

MyClass::MyClass() {
   cout << "Constructor called." << endl;
   dataObject = new int; // Allocate data object
   *dataObject = 0;
}

MyClass::~MyClass() {
   cout << "Destructor called." << endl;
   delete dataObject;
}

int main() {
   MyClass classObject1;
   MyClass classObject2;

   classObject1.SetDataObject(9);
   classObject2 = classObject1;
   cout << "classObject2: ";
   cout << classObject2.GetDataObject() << endl;

   return 0;
}
```




```
Constructor called.
Constructor called.
classObj2: 9
Destructor called.
Destructor called.
```

Animation content:

Static figure:
Begin Cpp code:

#include <iostream>
using namespace std;

class MyClass {
public:

```
  MyClass();
  ~MyClass();
  void SetDataObject(const int i) {*dataObject = i;}
  int GetDataObject() const {return *dataObject;}

private:
  int* dataObject;
};

MyClass::MyClass() {
  cout << "Constructor called." << endl;
  dataObject = new int; // Allocate data object
  *dataObject = 0;
}

MyClass::~MyClass() {
  cout << "Destructor called." << endl;
  delete dataObject;
}

int main() {
  MyClass classObject1;
  MyClass classObject2;

  classObject1.SetDataObject(9);
  classObject2 = classObject1;
  cout << "classObject2: ";
  cout << classObject2.GetDataObject() << endl;

  return 0;
}
```


End Cpp code.

Memory addresses 75 to 82 are shown. The variable, classObject1, is at memory address 75, contains the value 80, and is a dataObject. The variable, classObject2, is at memory address 78, contains the value 81, and is a dataObject. The value, 9, is at memory address 80, is labeled, classObject1's, and is an int. The value, 0, is at memory address 81, is labeled, classObject2's, and is an int. There is an arrow pointing to memory address 80 stating, Crash: already freed. There is an arrow pointing to memory address 81 stating, Memory leak. The output consosle has the lines:

```
Constructor called.
Constructor called.
classObj2: 9
Destructor called.
Destructor called.
```


Step 1: Two MyClass objects, classObject1 and classObject2, are created. classObject1's SetDataObject() function assigns the memory location pointed to by dataObject with 9.

The line of code, MyClass classObject1;, is highlighted. Memory address 75 contains the variable, classObject1, and is a dataObject. The line of code, MyClass::MyClass() {, is highlighted. The line of code, cout << "Constructor called." << endl, is highlighted. "Constructor called." is output to the

console. The line of code, dataObject = new int; // Allocate data object, is highlighted, and memory addres 80 is labeled, classObject1's, and is an int. The memory address 75 obtains the value 80. Memory address 80 obtains the value 0. The line of code, MyClass classObject2;, is highlighted. The variable, classObject2, is at memory address 78, contains the value 81, and is a dataObject. The value, 0, is at memory address 81, is labeled, classObject2's, and is an int. "Constructor called." is output to the console. The line of code, classObject1.SetDataObject(9);, is highlihgted, and the value at memory address 80 is updated to the value 9.

Step 2: The assignment classObject2 = classObject1; copies the pointer for classObject1's dataObject to classObject2, resulting in both dataObject members pointing to the same memory location.

The line of code, classObject2 = classObject1;, is highlighted, and the value, 80, is duplicated at memory address 75 and moves to memory address 78, where the value is updated to 80. The line of code, cout << "classObject2: ";, is highlighted, and classObject2: is output to the console. The line of code, cout << classObject2.GetDataObject() << endl;, is highlighed and the value, 9, at memory address 80 is output to the console.

Step 3: Destroying classObject1 frees that object's memory.

The line of code, return 0;, is highlighted. The line of code, MyClass::~MyClass() {, is highlighted. The line of code, cout << "Destructor called." << endl;, is highlighted, and Destructor called. is output to the console. The line of code, delete dataObject;, is highlighted and the information at memory address 75 is faded out. The value, 80, moves to memory address 80 and fades out memory address 80; labeling memory address 80, Freed.

Step 4: Destroying classObject2 then tries to free that same memory, causing a program crash. A memory leak also occurs because neither object is pointing to location 81.

The line of code, MyClass::~MyClass() {, is highlighted. The line of code, cout << "Destructor called." << endl;, is highlighted, and Destructor called. is output to the console. The line of code, delete dataObject;, is highlighted and the information at memory address 78 is faded out. The value, 80, moves to memory address 80 and is labeled Crash: Already freed. Memory address 81 is labeled Memory leak.

Animation captions:

1. Two MyClass objects, classObject1 and classObject2, are created. classObject1's SetDataObject() function assigns the memory location pointed to by dataObject with 9.
2. The assignment classObject2 = classObject1; copies the pointer for classObject1's dataObject to classObject2, resulting in both dataObject members pointing to the same memory location.
3. Destroying classObject1 frees that object's memory.
4. Destroying classObject2 then tries to free that same memory, causing a program crash. A memory leak also occurs because neither object is pointing to location 81.

PARTICIPATION ACTIVITY 8.10.2: Default assignment operator behavior.

1) The default assignment operator often works for objects without pointer members.
   - True
   - False

2) When used with objects with pointer members, the default assignment operator behavior may lead to crashes due to the same memory being freed more than once.
   - True
   - False

3) When used with objects with pointer members, the default assignment operator behavior may lead to memory leaks.
   - True
   - False

Overloading the assignment operator

The assignment operator (=) can be overloaded to eliminate problems caused by a memberwise assignment during an object copy. The implementation of the assignment operator iterates through each member of the source object. Each non-pointer member is copied directly from source member to destination member. For each pointer member, new memory is allocated, the source's referenced data is copied to the new memory, and a pointer to the new member is assigned as the destination member. Allocating and copying data for pointer members is known as a ***deep copy***.

The following program solves the default assignment operator behavior problem by introducing an assignment operator that performs a deep copy.

Figure 8.10.1: Assignment operator performs a deep copy.

```
#include <iostream>
using namespace std;

class MyClass {
public:
   MyClass();
   ~MyClass();
   MyClass& operator=(const MyClass& objToCopy);

   // Set member value dataObject
   void SetDataObject(const int setVal) {
      *dataObject = setVal;
   }

   // Return member value dataObject
   int GetDataObject() const {
      return *dataObject;
   }
private:
   int* dataObject;// Data member
};

// Default constructor
MyClass::MyClass() {
   cout << "Constructor called." << endl;
   dataObject = new int; // Allocate mem for data
   *dataObject = 0;
}

// Destructor
MyClass::~MyClass() {
   cout << "Destructor called." << endl;
   delete dataObject;
}

MyClass& MyClass::operator=(const MyClass& objToCopy) {
   cout << "Assignment op called." << endl;

   if (this != &objToCopy) {           // 1. Don't self-assign
      delete dataObject;               // 2. Delete old dataObject
      dataObject = new int;            // 3. Allocate new dataObject
      *dataObject = *(objToCopy.dataObject); // 4. Copy dataObject
   }

   return *this;
}

int main() {
   MyClass classObj1; // Create object of type MyClass
   MyClass classObj2; // Create object of type MyClass

   // Set and print object 1 data member value
   classObj1.SetDataObject(9);

   // Copy class object using copy assignment operator
   classObj2 = classObj1;

   // Set object 1 data member value
   classObj1.SetDataObject(1);

   // Print data values for each object
   cout << "classObj1:" << classObj1.GetDataObject() << endl;
   cout << "classObj2:" << classObj2.GetDataObject() << endl;

   return 0;
}
```


©zyBooks 01/31/24 17:50 1939727
Rob Daglio
MDCCOP2335Spring2024

```
Constructor called.
Constructor called.
Assignment op called.
obj1:1
obj2:9
Destructor called.
Destructor called.
```

PARTICIPATION ACTIVITY 8.10.3: Assignment operator.

1) Declare a copy assignment operator for a class named `EngineMap` using `inVal` as the input parameter name.

```
EngineMap& operator=(
                    );
```

Check Show answer

2) Provide the return statement for the copy assignment operator for the EngineMap class.

Check Show answer

CHALLENGE ACTIVITY 8.10.1: Enter the output of the program with an overloaded assignment operator.

539740.3879454.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

class SubstituteTeacher {
   public:
      SubstituteTeacher();
      ~SubstituteTeacher();
      SubstituteTeacher& operator=(const SubstituteTeacher& objToCopy);

      void SetSubject(const string setVal) {
         *subject = setVal;
      }

      string GetSubject() const {
         return *subject;
      }
   private:
      string* subject;
};

SubstituteTeacher::SubstituteTeacher() {
   subject = new string;
   *subject = "none";
}

SubstituteTeacher::~SubstituteTeacher() {
   delete subject;
}

SubstituteTeacher& SubstituteTeacher::operator=(const SubstituteTeacher& objToCopy) {
   if (this != &objToCopy) {
      delete subject;
      subject = new string;
      *subject = *(objToCopy.subject);
   }

   return *this;
}

int main() {
   SubstituteTeacher msDorf;
   SubstituteTeacher mrDiaz;
   SubstituteTeacher msPark;

   msPark.SetSubject("Bio");
   mrDiaz = msPark;
   msPark.SetSubject("Chem");
   msDorf = msPark;

   cout << msPark.GetSubject() << endl;
   cout << mrDiaz.GetSubject() << endl;

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 8.10.2: Copy assignment operator.

539740.3879454.qx3zqy7

Start

Complete the declaration of the copy assignment operator for Conference.

Ex: If the input is 3, then the output is:

```
Self-assignment not permitted
conference1: 17 guests
copyConference1: 3 guests
```

```
Destructor called
Destructor called
```

```
1  #include <iostream>
2  using namespace std;
3
4  class Conference {
5     public:
6        Conference();
7        ~Conference();
8        void setNumGuests(int newNumGuests);
9        void Print() const;
10       Conference& operator=(const Conference& conferenceToCopy);
11    private:
12       int* numGuests;
13 };
14
15 Conference::Conference() {
```


1 2 3

Check Next level

8.11 Rule of three

Classes have three special member functions that are commonly implemented together:

- **Destructor:** A destructor is a class member function that is automatically called when an object of the class is destroyed, such as when the object goes out of scope or is explicitly destroyed as in `delete someObject;`.
- **Copy constructor:** A copy constructor is another version of a constructor that can be called with a single pass by reference argument. The copy constructor is automatically called when an object is passed by value to a function, such as for the function `SomeFunction(MyClass localObject)` and the call `SomeFunction(anotherObject)`, when an object is initialized when declared such as `MyClass classObject1 = classObject2;`, or when an object is initialized when allocated via "new" as in `newObjectPtr = new MyClass(classObject2);`
- **Copy assignment operator:** The assignment operator "=" can be overloaded for a class via a member function, known as the copy assignment operator, that overloads the built-in function "operator=", the member function having a reference parameter of the class type and returning a reference to the class type.

The ***rule of three*** describes a practice that if a programmer explicitly defines any one of those three special member functions (destructor, copy constructor, copy assignment operator), then the programmer should explicitly define all three. For this reason, those three special member functions are sometimes called ***the big three***.

A good practice is to always follow the rule of three and define the big three if any one of these functions are defined.

PARTICIPATION ACTIVITY 8.11.1: Rule of three.

```
class MyClass {
public:
   MyClass();
```

Default constructor

```
    MyClass(int* someDataToCopy);

    ~MyClass();
    MyClass(const MyClass& objectToCopy);
    MyClass& operator=(const MyClass& objectToCopy);

    void MemberFunction1();
    void MemberFunction2();
    void MemberFunction3();
};
```

Other constructor (not a copy constructor)

The big three:
- Destructor
- Copy constructor
- Copy assignment operator

Animation content:

Static figure:
Begin Cpp code:

```
class MyClass {
public:
  MyClass();

  MyClass(int* someDataToCopy);


  ~MyClass();
  MyClass(const MyClass& objectToCopy);
  MyClass& operator=(const MyClass& objectToCopy);

  void MemberFunction1();
  void MemberFunction2();
  void MemberFunction3();
};
```

End Cpp code.

The line of code, MyClass();, is labeled Default constructor. The line of code, MyClass(int* someDataToCopy);, is labeled Other constructor (not a copy constructor). The lines of code,

~MyClass();
MyClass(const MyClass& objectToCopy);
MyClass& operator=(const MyClass& objectToCopy);,

are boxed and labeled The big three:
- Destructor
- Copy constructor
- Copy assignment operator.

Step 1: The big three consists of the destructor, copy constructor, and copy assignment operator.

The lines of code,

~MyClass();
MyClass(const MyClass& objectToCopy);
MyClass& operator=(const MyClass& objectToCopy);,

are boxed and labeled The big three:
- Destructor
- Copy constructor
- Copy assignment operator.

Step 2: The default constructor is not part of the big three.

The line of code, MyClass();, is labeled Default constructor.

Step 3: A constructor may exist that copies some data, but isn't the copy constructor. The copy constructor for MyClass takes a const MyClass& argument.

The line of code, MyClass(int* someDataToCopy);, is labeled Other constructor (not a copy constructor).

Animation captions:

1. The big three consists of the destructor, copy constructor, and copy assignment operator.
2. The default constructor is not part of the big three.
3. A constructor may exist that copies some data, but isn't the copy constructor. The copy constructor for MyClass takes a const MyClass& argument.

Default destructors, copy constructors, and assignment operators

- *If the programmer doesn't define a destructor for a class, the compiler implicitly defines one having no statements.*
- *If the programmer doesn't define a copy constructor for a class, then the compiler implicitly defines one whose statements do a memberwise copy, i.e.,* `classObject2.memberVal1 = classObject1.memberVal1`, `classObject2.memberVal2 = classObject1.memberVal2`, *etc.*
- *If the programmer doesn't define a copy assignment operator, the compiler implicitly defines one that does a memberwise copy.*

PARTICIPATION ACTIVITY 8.11.2: Rule of three.

1) If the programmer does not explicitly define a copy constructor for a class, copying objects of that class will not be possible.

- ○ True
- ○ False

2) The big three member functions for classes include a destructor, copy constructor, and default constructor.

- True
- False

3) If a programmer explicitly defines a destructor, copy constructor, or copy assignment operator, it is a good practice to define all three.

- True
- False

4) Assuming `MyClass prevObject` has already been declared, the statement `MyClass object2 = prevObject;` will call the copy assignment operator.

- True
- False

5) Assuming `MyClass prevObject` has already been declared, the following variable declaration will call the copy assignment operator.

```
MyClass object2;
...
object2 = prevObject;
```

- True
- False

Exploring further:

- More on Rule of Three in C++ from GeeksforGeeks.

8.12 C++ example: Employee list using vectors

zyDE 8.12.1: Managing an employee list using a vector.

The following program allows a user to add to and list entries from a vector, which maintains a list of employees.

1. Run the program, and provide input to add three employees' names and related data. Then use the list option to display the list.
2. Modify the program to implement the deleteEntry function.
3. Run the program again and add, list, delete, and list again various entries.

Load default template...

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// Add an employee
void AddEmployee(vector<string> &name, vector<string>
                 vector<string> &title) {
   string theName;
   string theDept;
   string theTitle;

   cout << endl << "Enter the name to add: " << endl;
   getline(cin, theName);
   cout << "Enter " << theName << "'s department: " <
```



```
a
Rajeev Gupta
Sales
```

Run

Below is a solution to the above problem.

zyDE 8.12.2: Managing an employee list using a vector (solution).

Load default template...

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;


// Add an employee
void AddEmployee(vector<string> &name, vector<string>
                 vector<string> &title) {
   string theName;
   string theDept;
   string theTitle;

   cout << endl << "Enter the name to add: " << endl;
   getline(cin, theName);
```



```
a
Rajeev Gupta
Sales
```

Run

8.13 LAB: Library book sorting

Two sorted lists have been created, one implemented using a linked list (**LinkedListLibrary** class) and the other implemented using the built-in Vector class (**VectorLibrary** class). Each list contains 100 books (title, ISBN number, author), sorted in ascending order by ISBN number.

Complete main() by inserting a new book into each list using the respective **LinkedListLibrary** and **VectorLibrary** InsertSorted() functions and outputting the number of book copy operations the computer must perform to insert the new book. Each InsertSorted() returns the number of book copy operations the computer performs.

Ex: If the input is:

```
Their Eyes Were Watching God
9780060838676
Zora Neale Hurston
```

the output is:


```
Number of linked list book copy operations: 1
Number of vector book copy operations: 95
```

Which list do you think will require the most operations? Why?

LAB ACTIVITY | 8.13.1: LAB: Library book sorting | 0 / 10

Current file: main.cpp Load default template...

```
#include "LinkedListLibrary.h"
#include "VectorLibrary.h"
#include "BookNode.h"
#include "Book.h"
#include <fstream>
#include <iostream>
using namespace std;

void FillLibraries(LinkedListLibrary &linkedListLibrary, VectorLibrary &vectc
   ifstream inputFS; // File input stream
   int linkedListOperations = 0;
   int vectorOperations = 0;

   BookNode* currNode;
   Book tempBook;
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

8.14 LAB: Mileage tracker for a runner

Given the `MileageTrackerNode` class containing the number of miles a runner runs on a certain date, complete main() to perform the following tasks:

1. Read the number of nodes in the linked list from user input, followed by the number of miles and date of each node.
2. Use the MileageTrackerNode's InsertAfter() function to insert each node according to the input sequence (i.e. insert each node after the last node).
3. Print the entire linked list using the MileageTrackerNode's PrintNodeData() function. **DO NOT print the head node that does not contain user-input values.** Hint: PrintNodeData() only outputs values in one node; therefore, iterate through

the linked list.

Ex. If the input is:

```
3
2.2
7/2/18
3.2
7/7/18
4.5
7/16/18
```


the output is:

```
2.2, 7/2/18
3.2, 7/7/18
4.5, 7/16/18
```


LAB ACTIVITY | 8.14.1: LAB: Mileage tracker for a runner | 0 / 10

Current file: main.cpp | Load default template...

```
1  #include "MileageTrackerNode.h"
2  #include <string>
3  #include <iostream>
4
5  using namespace std;
6
7  int main () {
8     // References for MileageTrackerNode objects
9     MileageTrackerNode* headNode;
10    MileageTrackerNode* currNode;
11    MileageTrackerNode* lastNode;
12
13    double miles;
14    string date;
15    int i;
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program | Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```


8.15 LAB: Inventory (linked lists: insert at the front of a list)

Given main(), define an InsertAtFront() member function in the `InventoryNode` class that inserts items at the front of a linked list (after the head node, which not contain user-input values).

Ex. If the input is:

```
4
plates 100
spoons 200
cups 150
forks 200
```

the output is:

```
200 forks
150 cups
200 spoons
100 plates
```

Hint: To determine the parameters for InsertAtFront(), check the function call in main().

LAB ACTIVITY | 8.15.1: LAB: Inventory (linked lists: insert at the front of a list) 0 / 10

File is marked as read only Current file: main.cpp

```
1  #include "InventoryNode.h"
2
3  int main() {
4     int count;
5     int numItems;
6     string item;
7
8     InventoryNode *headNode = new InventoryNode();
9     InventoryNode *currNode;
10
11    // Obtain number of items
12    cin >> count;
13
14    // Get each item and number of each
15    for (int i = 0; i < count; i++) {
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

8.16 LAB: Playlist (output linked list)

Given main(), complete the `SongNode` class to include the function PrintSongInfo(). Then write the PrintPlaylist() function in main.cpp to print all songs in the playlist. **DO NOT print the head node, which does not contain user-input values.**

Ex: If the input is:

```
Stomp!
380
The Brothers Johnson
The Dude
337
Quincy Jones
You Don't Own Me
151
Lesley Gore
-1
```

the output is:


```
LIST OF SONGS
-------------
Title: Stomp!
Length: 380
Artist: The Brothers Johnson

Title: The Dude
Length: 337
```

```
Artist: Quincy Jones

Title: You Don't Own Me
Length: 151
Artist: Lesley Gore
```

539740.3879454.qx3zqy7

LAB ACTIVITY | 8.16.1: LAB: Playlist (output linked list) 0 / 10

Current file: main.cpp Load default template...

```
1  #include <iostream>
2  #include "SongNode.h"
3
4  // TODO: Write PrintPlaylist() function
5
6  int main() {
7     SongNode* headNode;
8     SongNode* currNode;
9     SongNode* lastNode;
10
11    string songTitle;
12    string songLength;
13    string songArtist;
14
15    // Front of nodes list
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

8.17 LAB: Grocery shopping list (linked list: inserting at the end of a list)

Given main(), define an InsertAtEnd() member function in the `ItemNode` class that adds an element to the end of a linked list. **DO NOT print the head node that does not contain user-input values.**

Ex. if the input is:


```
4
Kale
Lettuce
Carrots
Peanuts
```

where 4 is the number of items to be inserted; Kale, Lettuce, Carrots, Peanuts are the names of the items to be added at the end of the list.

The output is:

```
Kale
Lettuce
Carrots
Peanuts
```


LAB ACTIVITY | 8.17.1: LAB: Grocery shopping list (linked list: inserting at the end of a list) 0 / 10

File is marked as read only Current file: main.cpp

```
1  #include "ItemNode.h"
2
3  int main() {
4     ItemNode *headNode;  // Create intNode objects
5     ItemNode *currNode;
6     ItemNode *lastNode;
7
8     string item;
9     int i;
10    int input;
11
12    // Front of nodes list
13    headNode = new ItemNode();
14    lastNode = headNode;
15
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)