4.1 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.1.1: Dynamically allocated arrays.

Animation content:

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 4.1.2: Dynamically allocated arrays.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.1.3: Vector insert performance problem.

Animation content:

undefined

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 4.1.4: A list avoids the shifting problem.

Animation content:

undefined

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.1.1: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.1.5: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.1.6: Pointers used to call class member functions.

Animation content:

undefined

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".

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.1.7: 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;

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Exploring further:

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

4.2 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 4.2.1: Assigning a pointer with an address.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.2.2: 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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.2.3: Using the dereference operator.

Animation content:

undefined

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 4.2.4: Dereferencing a pointer.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.2.5: Checking to see if a pointer is null.

Animation content:

undefined

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.2.6: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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.*

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.2.7: Common pointer errors.

Animation content:

undefined

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 4.2.8: Common pointer errors.

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

1)

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- ○ 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;
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

zyDE 4.2.1: 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 o statements will print the same output. Then run and observe the output. Did you pre correctly?

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Load default template... Run

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

CHALLENGE ACTIVITY 4.2.1: Enter the output of pointer content.

489394.3384924.qx3zqy7

Start

Type the program's output

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

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

   someNumber = 10;
   numberPointer = &someNumber;

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

   return 0;
}
```

1 2

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check Next

CHALLENGE ACTIVITY 4.2.2: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

489394.3384924.qx3zqy7

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

int main() {
   int* numItemsPointer;
   int numItems;

   cin >> numItems;

   /* Your solution goes here  */

   if (numItemsPointer == nullptr) {
      cout << "Items is negative" << endl;
   }
   else {
```

Run

View your last submission

CHALLENGE ACTIVITY | 4.2.3: Pointer basics.

489394.3384924.qx3zqy7

Start

Given variables level, time, and alert, declare and assign the following pointers:

- double pointer levelPointer is assigned with the address of level.
- integer pointer timePointer is assigned with the address of time.
- character pointer alertPointer is assigned with the address of alert.

Ex: If the input is `7.5 19 H`, then the output is:

```
Tide level: 7.5 meters
Recorded at hour: 19
Alert: H
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

int main() {
   double level;
   int time;
   char alert;
```

```
10    /* Your code goes here */
11
12    cin >> level;
13    cin >> time;
14    cin >> alert;
15
```

1 2

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check Next level

4.3 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 4.3.1: The new operator allocates space for an object, then calls the constructor.

Animation content:

undefined

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 4.3.2: The new operator.

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.3.3: Constructor arguments.

Animation content:

undefined

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 4.3.4: Constructor arguments.

If unable to drag and drop, refresh the page.

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

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

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

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

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

Reset

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.3.1: 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();`

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.3.5: 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`

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.3.6: The delete operator.

Animation content:

undefined

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.3.7: 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 4.3.8: Allocating and deleting an array of Point objects.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

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 4.3.9: Allocating and deleting object arrays.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.3.1: Using the new, delete, and -> operators.

489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#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(55);
   myCar2 = new Car(60);

   return 0;
}
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2

Check Next

CHALLENGE ACTIVITY 4.3.2: 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

489394.3384924.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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

View your last submission

CHALLENGE ACTIVITY 4.3.3: Operators: new, delete, and ->.

489394.3384924.qx3zqy7

Start

Two integers are read as the velocity and the duration of a MovingBody object. Assign pointer myMovi
MovingBody object using the velocity and the duration as arguments in that order.

Ex: If the input is `8 13`, then the output is:

```
MovingBody's velocity: 8
MovingBody's duration: 13
```

```
1  #include <iostream>
2  using namespace std;
3
4  class MovingBody {
5     public:
6        MovingBody(int velocityValue, int durationValue);
7        void Print();
8     private:
9        int velocity;
10       int duration;
11 };
12 MovingBody::MovingBody(int velocityValue, int durationValue) {
13    velocity = velocityValue;
14    duration = durationValue;
15 }
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2 3 4

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

4.4 String functions with pointers

C string library functions

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

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

Animation content:

undefined

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.

©zyBooks 05/30/23 21:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.4.2: 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

©zyBooks 05/30/23 21:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023

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 4.4.1: 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."`
		`}`

©zyBooks 05/30/23 21:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.4.3: 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"

©zyBooks 05/30/23 21:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.4.1: String searching example.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#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;
}
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
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 4.4.4: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY 4.4.1: Enter the output of the string functions.

489394.3384924.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. Ron Smith");

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

   return 0;
}
```

1 2 3 4

Check Next

CHALLENGE ACTIVITY 4.4.2: Find char in C string

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

Learn how our autograder works

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

489394.3384924.qx3zqy7

```
1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 int main() {
6    char personName[100];
7    char searchChar;
8    char* searchResult = nullptr;
```

```
 9
10    cin.getline(personName, 100);
11    cin >> searchChar;
12
13    /* Your solution goes here  */
14
15    if (searchResult != nullptr) {
```

Run

View your last submission

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY 4.4.3: Find C string in C string.

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

Learn how our autograder works

489394.3384924.qx3zqy7

```
 1 #include <iostream>
 2 #include <cstring>
 3 using namespace std;
 4
 5 int main() {
 6    char movieTitle[100];
 7    char* movieResult = nullptr;
 8
 9    cin.getline(movieTitle, 100);
10
11    /* Your solution goes here  */
12
13    cout << "Movie title contains The? ";
14    if (movieResult != nullptr) {
15       cout << "Yes." << endl;
```

Run

View your last submission

4.5 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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea

Figure 4.5.1: A basic example to introduce linked lists.

```
-1
555
777
999
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#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
objects
   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();
   }
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.5.1: Inserting nodes into a basic linked list.

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.5.2: 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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#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 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

zyDE 4.5.1: Managing a linked list.

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

Load default template... Run

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

class IntNode {
public:
   IntNode(int dataInit = 0, IntNod
   void InsertAfter(IntNode* nodeLo
   IntNode* GetNext();
   void PrintNodeData();
   int GetDataVal();
private:
   int dataVal;
   IntNode* nextNodePtr;
};
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.5.1: Enter the output of the program using Linked List.

489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#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("Pavanne");
   headObj->InsertAfter(firstSong);

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

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

   currObj = headObj;

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2

Check Next

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY 4.5.2: A first linked list.

489394.3384924.qx3zqy7

Start

Two integers, neighbors1 and neighbors2, are read from input as the number of neighbors of two towr
value of -1. Create a new node firstTown with integer neighbors1 and insert firstTown after headObj. Tl
secondTown with integer neighbors2 and insert secondTown after firstTown.

Ex: If the input is `24 11`, then the output is:

```
-1
24
11
```

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

class TownNode {
   public:
      TownNode(int neighborsInit = 0, TownNode* nextLoc = nullptr);
      void InsertAfter(TownNode* nodeLoc);
      TownNode* GetNext();
      void PrintNodeData();
   private:
      int neighborsVal;
      TownNode* nextNodePtr;
};

TownNode::TownNode(int neighborsInit, TownNode* nextLoc) {
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2

Check Next level

4.6 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 4.6.1: Use of the four memory regions.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.6.2: Stack and heap definitions.

If unable to drag and drop, refresh the page.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

	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

4.7 Memory leaks

Memory leak

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.7.1: Memory leak can use up all available memory.

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.7.2: Memory leaks.

If unable to drag and drop, refresh the page.

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

	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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.7.3: Lack of destructor yields memory leak.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

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 4.7.4: 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;`

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.7.5: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

4.8 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 4.8.1: LinkedList nodes are not deallocated without a LinkedList class destructor.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

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 4.8.2: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.8.1: 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;
}
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.8.3: The LinkedList class destructor, called when the list is deleted, frees all nodes.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

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 4.8.4: LinkedList class destructor.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.8.5: Destructors are called automatically only for non-reference/pointer variables.

Animation content:

undefined

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.8.6: When a destructor is called.

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

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

- True
- False

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.8.1: Enter the output of the destructors.

489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#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(4);
   IntNode* node3 = new IntNode(6);
   IntNode* node4 = new IntNode(7);

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

   return 0;
}
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2

Check Next

CHALLENGE ACTIVITY 4.8.2: Destructors.

489394.3384924.qx3zqy7

Start

Complete the TownNode class destructor. The destructor prints "Deallocating TownNode (", followed b
a space, and the value of population, and then ")". End with a newline.

Ex: If the input is Opal 4106, then the output is:

```
Deallocating TownNode (Opal 4106)
```

```
1  #include <iostream>
2  using namespace std;
3
4  class TownNode {
5     public:
6        TownNode(string nameValue, int populationValue) {
7           name = nameValue;
8           population = populationValue;
9        }
10
11       /* Your code goes here */
12
13       string name;
14       int population;
15       TownNode* next;
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2

Check Next level

Exploring further:

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

4.9 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 4.9.1: Copying an object without a copy constructor.

Animation content:

undefined

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 4.9.2: 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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.9.1: 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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 4.9.1: 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.
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.9.3: 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);`

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

4) `EncBlock vidBlock = myBlock;`
   - `EncBlock();`
   - `EncBlock(const EncBlock& origObj);`
   - `EncBlock(int blockSize);`

Exploring further:

- More on Copy Constructors from cplusplus.com

CHALLENGE ACTIVITY 4.9.1: Enter the output of the copy constructors.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

489394.3384924.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);

   node3 = node2;
   node2.SetNumVal(9);

   cout << node3.GetNumVal() << " " << node2.GetNumVal() << endl;

   return 0;
}
```

1 2

Check Next

CHALLENGE ACTIVITY 4.9.2: Write a copy constructor.

Write a copy constructor for CarCounter that assigns origCarCounter.carCount to the constructed object's carCount. Sample output for the given program:

```
Cars counted: 5
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Learn how our autograder works

489394.3384924.qx3zqy7

```
1 #include <iostream>
2 using namespace std;
3
4 class CarCounter {
```

```
5     public:
6        CarCounter();
7        CarCounter(const CarCounter& origCarCounter);
8        void SetCarCount(const int count) {
9           carCount = count;
10       }
11       int GetCarCount() const {
12          return carCount;
13       }
14    private:
15       int carCount;
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Run

View your last submission

CHALLENGE ACTIVITY | 4.9.3: Copy constructors.

489394.3384924.qx3zqy7

Start

SavingsAccount is a class with two double* data members pointing to the amount saved and interest respectively. Two doubles are read from input to initialize userAccount. Use the copy constructor to cre object named copyAccount that is a deep copy of userAccount.

Ex: If the input is 80.00 0.03, then the output is:

```
Original constructor called
Called SavingsAccount's copy constructor
userAccount: $80.00 with 3.00% interest rate
copyAccount: $160.00 with 6.00% interest rate
```

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  class SavingsAccount {
6     public:
7        SavingsAccount(double startingSaved = 0.0, double startingRate = 0.0);
8        SavingsAccount(const SavingsAccount& acc);
9        void SetSaved(double newSaved);
10       void SetRate(double newRate);
11       double GetSaved() const;
12       double GetRate() const;
13       void Print() const;
14    private:
15       double* saved;
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2

Check Next level

4.10 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:

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
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 4.10.1: Basic assignment operation fails when pointer member involved.

Animation content:

undefined

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 4.10.2: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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***.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

Figure 4.10.1: Assignment operator performs a deep copy.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#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 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

PARTICIPATION ACTIVITY 4.10.3: Assignment operator.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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 4.10.1: Enter the output of the program with an overloaded assignment operator.

489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#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("English");
   mrDiaz = msPark;
   mrDiaz.SetSubject("Art");
   msDorf = mrDiaz;

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

   return 0;
}
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2

Check Next

CHALLENGE ACTIVITY 4.10.2: Copy assignment operator.

489394.3384924.qx3zqy7

Start

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Complete the copy assignment operator to prevent self-assignment.

Ex: If the input is 3.30, then the output is:

```
Self-assignment not permitted
account1: 1.70% rate
copyAccount1: 3.30% rate
```

```
Destructor called
Destructor called
```

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

class SavingsAccount {
   public:
      SavingsAccount();
      ~SavingsAccount();
      void setRatePercent(double newRatePercent);
      void Print() const;
      SavingsAccount& operator=(const SavingsAccount& accountToCopy);
   private:
      double* ratePercent;
};
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1 2

Check Next level

4.11 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.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY 4.11.1: Rule of three.

Animation content:

undefined

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- *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 4.11.2: 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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

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

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Exploring further:

- More on Rule of Three in C++ from GeeksforGeeks.

4.12 C++ example: Employee list using vectors

zyDE 4.12.1: Managing an employee list using a vector.

The following program allows a user to add to and list entries from a vector, which maint list of employees.

1. Run the program, and provide input to add three employees' names and related dat 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 templat

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

// Add an employee
void AddEmployee(vector<string> &name, vector<string> &department,
                 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: " << endl;
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
a
Rajeev Gupta
Sales
```

Run

Below is a solution to the above problem.

zyDE 4.12.2: Managing an employee list using a vector (solution).

Load default templat

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


// Add an employee
void AddEmployee(vector<string> &name, vector<string> &department,
                 vector<string> &title) {
   string theName;
   string theDept;
   string theTitle;

   cout << endl << "Enter the name to add: " << endl;
   getline(cin, theName);
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
a
Rajeev Gupta
Sales
```

Run

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023