10.1 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 10.1.1: 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 10.1.2: 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 10.1.3: 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 10.1.4: 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.

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 10.1.1: 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 10.1.5: 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 10.1.6: 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".

PARTICIPATION ACTIVITY 10.1.7: 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

10.2 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 10.2.1: Assigning a pointer with an address.

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.

PARTICIPATION ACTIVITY 10.2.2: 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 10.2.3: 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 10.2.4: 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 10.2.5: 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.

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 assign a null pointer.

PARTICIPATION ACTIVITY 10.2.6: 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: `*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 10.2.7: 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 10.2.8: 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 10.2.1: 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 statements will print the same output. Then run and observe the output. Did you p correctly?

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 the r
11                        // POINTED TO by
12
13    // vehicleMpg = 40.0;    // Uncomment
14
15    cout << "Vehicle MPG = " << vehicleM
16    cout << "Vehicle MPG = " << *valPoir
17
```


CHALLENGE ACTIVITY 10.2.1: Enter the output of pointer content.

Start

Type the program's output

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

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

   someNumber = 5;
   numberPointer = &someNumber;

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

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 10.2.2: 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
```

```
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 {
16       cout << "Items: " << *numItemsPointer << endl;
17    }
```

Run

View your last submission

10.3 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 | 10.3.1: 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 | 10.3.2: 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 10.3.3: 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 10.3.4: Constructor arguments.

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

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

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

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


	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 10.3.1: 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 10.3.5: 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 10.3.6: The delete operator.

Animation content:

`undefined`

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 10.3.7: The delete operator.

1) The delete operator can be used on 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 10.3.8: Allocating and deleting an array of Point objects.

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 10.3.9: 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 10.3.1: Operators: new, delete, and ->.

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(70);
   myCar2 = new Car(80);

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 10.3.2: Deallocating memory

Deallocate memory for kitchenPaint using the delete operator.

```
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;
16
17    kitchenPaint = new PaintContainer;
```

Run

View your last submission

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

©zyBooks 04/25/21 07:06 488201
xiang zhao
BAYLORCSI14301440Spring2021

10.4 Array-based lists

Array-based lists

An ***array-based list*** is a list ADT implemented using an array. An array-based list supports the common list ADT operations, such as append, prepend, insert after, remove, and search.

In many programming languages, arrays have a fixed size. An array-based list implementation will dynamically allocate the array as needed as the number of elements changes. Initially, the array-based list implementation allocates a fixed size array and use a length variable to keep track of how many array elements are in use. The list starts with a default allocation size, greater than or equal to 1. A default size of 1 to 10 is common.

Given a new element, the ***append*** operation for an array-based list of length X inserts the new element at the end of the list, or at index X.

PARTICIPATION ACTIVITY 10.4.1: Appending to array-based lists.

Animation captions:

1. An array of length 4 is initialized for an empty list. Variables store the allocation size of 4 and list length of 0.
2. Appending 45 uses the first entry in the array, and the length is incremented to 1.
3. Appending 84, 12, and 78 uses the remaining space in the array.

PARTICIPATION ACTIVITY 10.4.2: Array-based lists.

1) The length of an array-based list equals the list's array allocation size.
   - ○ True
   - ○ False

2) An item can be appended to an array-based list, provided the length is less than the array's allocated size.
   - ○ True
   - ○ False

3) An array-based list can have a default allocation size of 0.
   - ○

- True
- False

Resize operation

An array-based list must be resized if an item is added when the allocation size equals the list length. A new array is allocated with a length greater than the existing array. Allocating the new array with twice the current length is a common approach. The existing array elements are then copied to the new array, which becomes the list's storage array.

Because all existing elements must be copied from 1 array to another, the resize operation has a runtime complexity of O(N).

PARTICIPATION ACTIVITY 10.4.3: Array-based list resize operation.

Animation content:

undefined

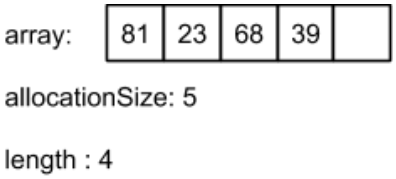
Animation captions:

1. The allocation size and length of the list are both 4. An append operation cannot add to the existing array.
2. To resize, a new array is allocated of size 8, and the existing elements are copied to the new array. The new array replaces the list's array.
3. 51 can now be appended to the array.

PARTICIPATION ACTIVITY 10.4.4: Array-based list resize operation.

Assume the following operations are executed on the list shown below:

ArrayListAppend(list, 98)
ArrayListAppend(list, 42)
ArrayListAppend(list, 63)



1) Which operation causes ArrayListResize to be called?

- ArrayListAppend(list, 98)
- ArrayListAppend(list, 42)
- ArrayListAppend(list, 63)

2) What is the list's length after 63 is appended?

- 5
- 7
- 10

3) What is the list's allocation size after 63 is appended?

- 5
- 7
- 10

Prepend and insert after operations

The ***Prepend*** operation for an array-based list inserts a new item at the start of the list. First, if the allocation size equals the list length, the array is resized. Then all existing array elements are moved up by 1 position, and the new item is inserted at the list start, or index 0. Because all existing array elements must be moved up by 1, the prepend operation has a runtime complexity of O(N).

The ***InsertAfter*** operation for an array-based list inserts a new item after a specified index. Ex: If the contents of `numbersList` is: 5, 8, 2, `ArrayListInsertAfter(numbersList, 1, 7)` produces: 5, 8, 7, 2. First, if the allocation size equals the list length, the array is resized. Next, all elements in the array residing after the specified index are moved up by 1 position. Then, the new item is inserted at index (specified index + 1) in the list's array. The InsertAfter operation has a best case runtime complexity of O(1) and a worst case runtime complexity of O(N).

InsertAt operation.

Array-based lists often support the InsertAt operation, which inserts an item at a specified index. Inserting an item at a desired index X can be achieved by using InsertAfter to insert after index X - 1.

PARTICIPATION ACTIVITY 10.4.5: Array-based list prepend and insert after operations.

Animation content:

undefined

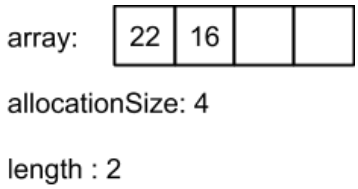
Animation captions:

1. To prepend 91, every array element is first moved up 1 index.
2. Item 91 is assigned at index 0.
3. Inserting item 36 after index 2 requires elements at indices 3 and higher to be moved up 1. Item 36 is inserted at index 3.

PARTICIPATION ACTIVITY 10.4.6: Array-based list prepend and insert after operations.

Assume the following operations are executed on the list shown below:

ArrayListPrepend(list, 76)
ArrayListInsertAfter(list, 1, 38)
ArrayListInsertAfter(list, 3, 91)



1) Which operation causes ArrayListResize to be called?
   - ArrayListPrepend(list, 76)
   - ArrayListInsertAfter(list, 1, 38)
   - ArrayListInsertAfter(list, 3, 91)

2) What is the list's allocation size after all operations have completed?
   - 5
   - 8
   - 10

3) What are the list's contents after all operations have completed?
   - 22, 16, 76, 38, 91
   - 76, 38, 22, 91, 16
   - 76, 22, 38, 16, 91

Search and removal operations

Given a key, the ***search*** operation returns the index for the first element whose data matches that key, or -1 if not found.

Given the index of an item in an array-based list, the ***remove-at*** operation removes the item at that index. When removing an item at index X, each item after index X is moved down by 1 position.

Both the search and remove operations have a worst case runtime complexity of $O(N)$.

PARTICIPATION ACTIVITY 10.4.7: Array-based list search and remove-at operations.

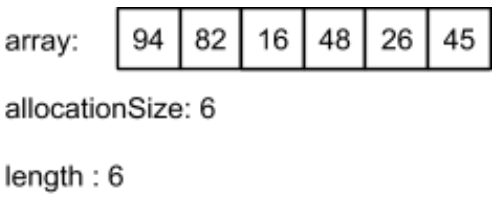
Animation content:

undefined

Animation captions:

1. The search for 84 compares against 3 elements before returning 2.
2. Removing the element at index 1 causes all elements after index 1 to be moved down to a lower index.
3. Decreasing the length by 1 effectively removes the last 51.
4. The search for 84 now returns 1.

PARTICIPATION ACTIVITY 10.4.8: Search and remove-at operations.



1) What is the return value from ArrayListSearch(list, 33)?

Check Show answer

2) When searching for 48, how many elements in the list will be compared with 48?

Check Show answer

3) ArrayListRemoveAt(list, 3) causes how many items to be moved down by 1 index?

Check Show answer

4) ArrayListRemoveAt(list, 5) causes how many items to be moved down by 1 index?

Check Show answer

PARTICIPATION ACTIVITY 10.4.9: Search and remove-at operations.

1) Removing at index 0 yields the best case runtime for remove-at.
   - True
   - False

2) Searching for a key that is not in the list yields the worst case runtime for search.
   - True
   - False

3) Neither search nor remove-at will resize the list's array.
   - True
   - False

CHALLENGE ACTIVITY 10.4.1: Array-based lists.

Start

numList: | 96 | 19 | |

allocationSize: 3
length: 2

If an item is added when the allocation size equals the array length, a new array with twice the current length is allocated.

Determine the length and allocation size of numList after each operation.

Operation	Length	Allocation size
ArrayListAppend(numList, 58)	Ex: 1	Ex:1
ArrayListAppend(numList, 89)		
ArrayListAppend(numList, 95)		
ArrayListAppend(numList, 63)		
ArrayListAppend(numList, 25)		

1 2 3 4

Check Next

10.5 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 10.5.1: 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.

PARTICIPATION ACTIVITY 10.5.2: 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 10.5.1: 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 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 10.5.3: 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 10.5.1: 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 10.5.4: 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 10.5.1: Enter the output of the string functions.

Start

Type the program's output

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

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

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

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

   return 0;
}
```


1 2 3 4

Check Next

CHALLENGE ACTIVITY 10.5.2: Find char in C string

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


```
#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) {
      cout << "Character found." << endl;
   }
   else {
```

Run

CHALLENGE ACTIVITY 10.5.3: Find C string in C string.

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


```
#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;
   }
   else {
```

Run

10.6 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 10.6.1: Use of the four memory regions.

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 10.6.2: Stack and heap definitions.

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

	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

10.7 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 10.7.1: A basic example to introduce linked lists.

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

class IntNode {
public:
   IntNode(int dataInit = 0, IntNode* nextLoc = nullptr);
   void InsertAfter(IntNode* nodeLoc);
   IntNode* GetNext();
```

```
-1
555
777
999
```

```
   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);
   nodeObj1->InsertAfter(nodeObj3);

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

   return 0;
}
```


©zyBooks 04/25/21 07:06 488201
xiang zhao
BAYLORCSI14301440Spring2021

PARTICIPATION

ACTIVITY 10.7.1: 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 exists, 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 10.7.2: 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 10.7.2: Managing many new items using just a few pointer variables.

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

```
#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 objects
   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;
}
```

zyDE 10.7.1: 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, IntNode* r
   void InsertAfter(IntNode* nodeLoc);
   IntNode* GetNext();
   void PrintNodeData();
   int GetDataVal();
private:
   int dataVal;
   IntNode* nextNodePtr;
};

// Constructor
```

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

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

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

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

   currObj = headObj;

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

1 2

Check Next

CHALLENGE ACTIVITY 10.7.2: Linked list negative values counting.

Assign negativeCntr with the number of negative values in the linked list.

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

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

// Constructor
IntNode::IntNode(int dataInit, IntNode* nextLoc) {
```

Run

View your last submission

10.8 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 | 10.8.1: LinkedList nodes are not deallocated without a LinkedList class destructor.

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 | 10.8.2: 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

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 10.8.1: 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;
}
```


PARTICIPATION ACTIVITY 10.8.3: The LinkedList class destructor, called when the list is deleted, frees all nodes.

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 10.8.4: 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 10.8.5: Destructors are called automatically only for non-reference/pointer variables.

Animation content:

`undefined`

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 10.8.6: 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 10.8.1: Enter the output of the destructors.

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(2);
   IntNode* node2 = new IntNode(3);
   IntNode* node3 = new IntNode(5);
   IntNode* node4 = new IntNode(7);

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

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 10.8.2: Write a destructor.

Write a destructor for the CarCounter class that outputs the following. End with newline.

```
Destroying CarCounter
```



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

```
6       CarCounter();
7       ~CarCounter();
8    private:
9       int carCount;
10 };
11
12 CarCounter::CarCounter() {
13    carCount = 0;
14 }
15
16 /* Your solution goes here  */
17
```


Run

Exploring further:

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

10.9 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 10.9.1: 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.

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 | 10.9.2: Memory leaks.

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 10.9.3: Lack of destructor yields memory leak.

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 10.9.4: 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 10.9.5: 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

10.10 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 10.10.1: 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 10.10.2: 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 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 10.10.1: 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 10.10.1: 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 10.10.3: 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 10.10.1: Enter the output of the copy constructors.

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(7);

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

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 10.10.2: 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
```

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

class CarCounter {
   public:
      CarCounter();
      CarCounter(const CarCounter& origCarCounter);
      void SetCarCount(const int count) {
         carCount = count;
      }
      int GetCarCount() const {
         return carCount;
      }
   private:
      int carCount;
};

CarCounter::CarCounter() {
```

Run

10.11 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 10.11.1: 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 10.11.2: 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 10.11.1: 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 04/25/21 07:06 488201
xiang zhao
BAYLORCSI14301440Spring2021

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


PARTICIPATION ACTIVITY 10.11.3: 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 10.11.1: Enter the output of the program with an overloaded assignment operator.

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

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

   return 0;
}
```


1 2

©zyBooks 04/25/21 07:06 488201
xiang zhao
BAYLORCSI14301440Spring2021

Check Next

CHALLENGE ACTIVITY 10.11.2: Write a copy assignment.

Write a copy assignment operator for CarCounter that assigns objToCopy.carCount to the new objects' carCount, then returns *this. Sample output for the given program:

```
Cars counted: 12
```



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

class CarCounter {
   public:
      CarCounter();
      ~CarCounter();
      CarCounter& operator=(const CarCounter& objToCopy);
      void SetCarCount(const int setVal) {
         *carCount = setVal;
      }
      int GetCarCount() const {
         return *carCount;
      }
   private:
      int* carCount;
};
```

Run

10.12 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 10.12.1: 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

- *If the programmer doesn't define a destructor for a class, the compiler implicitly defines one having no statements, meaning the destructor does nothing.*
- *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 10.12.2: 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.

10.13 C++ example: Employee list using vectors

zyDE 10.13.1: Managing an employee list using a vector.

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

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

```
#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;
   getline(cin, theDept);
   cout << "Enter " << theName << "'s title: " << endl;
```

```
a
Rajeev Gupta
Sales
```

Run

Below is a solution to the above problem.

zyDE 10.13.2: Managing an employee list using a vector (solution).

Load default templ

```
#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;
   getline(cin, theDept);
```



```
a
Rajeev Gupta
Sales
```

Run

10.14 Lab 1 - Pointer Introduction

Pointers can be considered the most confusing concept tackled in this course. Mastery of pointers is not only essential for success in this class, but is also essential for success in subsequent classes. The goal of this lab is to introduce you to pointers and a tool that can prove indispensable for developing software solutions that use pointers. I am not going to force you to use CLion - I am recommending it -, but I am requiring each of you to learn to use your choice of IDE's debugger and learn to manage multiple files with main() functions within your IDE.

For this lab, we will not use zyBooks' testing functionality. All testing will come from your local computer.

Your code should produce errors stating that there are multiple main() functions only on zyBooks. Your code should work on the your local computer. If you get the error stating multiple main() functions, you will need to remove all files except the current file from you project.

First, we need to discuss the requirements for this lab. You will need to use the file named lab1-answersToQuestions.h to submit your answers to questions found in this lab to zyBooks. To get full credit, you will need to start the .h file off by including the following information. I have provided a template that is intended to provided you with space to answer the questions. The entire file should be comments. Make sure that you update the file with your information.

Name

CSI 1440 Section X (where X is your section number)

Lab 1

Pointer introduction

As we are learning, pointers are variables that store addresses. But, how do we get the address of a variable? We can request the address of a variable using the reference operator (&).

Create a new project, named Pointers and remove the default main.cpp from the project. Add a new file named “pointers1.cpp”. Make sure you write the proper comments for the file using “pointers1.cpp” as the filename and “Testing the reference operator” as the description for the file. Also, add comments within the code to explain what is going on.

```
/**
-  file: pointers1.cpp
-  author:  Prof. Matthew Aars
-  course: CSI 1440
-  assignment: Lab 0
-  due date: 8/25/2016
-
-  Date Modified: 7/9/2020
-     - Lab changed for zyBooks
-  Date Modified: 8/25/2014
-    - File Created
-
-  Testing the reference operator
*/
#include <iostream>

using namespace std;

int main() {
```

```
  int x = 5;

    cout << &x << " is the address of x" << endl;
    cout << x << " is the value stored in x" << endl;

    return 0;
}
```


Compile and run your code. Then, submit your code below.

The Reference Operator

Remove the previous file from your project and add a new file named "pointers1-1.cpp" to the project. Make sure you write the proper comments for the file using "pointers1-1.cpp" as the name of the file and "The Reference Operator" as the description for the file. Also, add comments within the code to explain what is going on.

```
#include <iostream>

using namespace std;

int main() {
    int x;
    int *intPtr;

    x = 10;
    intPtr = &x;

    cout << "x's value is " << x << endl;
    cout << "intPtr's value is " << intPtr << endl;

    cout << "x's address is " << &x << endl;

    cout << "The size of x is " << sizeof(x) << endl;
    cout << "The size of intPtr is " << sizeof(intPtr) << endl;

    return 0;
}
```


Compile and run your code.

Edit the code for "pointers1-1.cpp" by adding a char named "ch" and a char * named "chPtr". Print out the same information that was printed for the integers. (You will most likely need to typecast "chPtr" as an integer pointer to print out the address)

Compile and run your code. Then, submit your code below.

Using the Debugger (These instructions are intended for use with CLion 2019)

Before debugging, you must set a breakpoint in your program by clicking the area to the right of your code. It will display a red dot when you have successfully set a breakpoint. You can remove the breakpoint by clicking on the red dot. Breakpoints are used to inform the debugger to pause execution at that point which will allow you to see the value of all the variables in your program at that time of execution. You can set multiple breakpoints in your program, but for this exercise lets keep to just one.

Place a breakpoint on the line assigning a value of 10 to x. Now start the Debugger by pressing the green bug button found just to the right of the green play button or by selecting Debug … in the Run menu in the Menu Bar.

Your code should build (compile) then begin execution. Instead of executing all the way through your code as it has done in the past. It should pause execution at the line in which you placed the breakpoint. At this point you should be able to view the contents of all the variables in your code by looking at a window labeled Variables. This widow also let's you set up Watches (specific variables you wish to look at) and allows for viewing of Locals (all variables declared in the current scope level).

Use the step over button or step over selection in the Run menu to move to the next line in the code. Pay attention to the values of your variables as you run through the entire code one line at a time.

Make sure that you understand how to use your debugger. It will cut countless hours off of development time when you encounter runtime errors.

Pointers are variables too

Just like the variables we have our pointers pointing to, pointers have addresses too.

Remove the previous file from the project and add a new file named "pointers1-2.cpp" to the project. Write a program that declares a double pointer named "dblPtr" and a double named "dbl". Then, implement the following instructions. Make sure you write the proper comments for the file using "pointers1-2.cpp" as the name of the file and "Pointer as a variable" as the description for the file. Also, add comments within the code to explain what is going on.

1. Assign a value of 2.5 to "dbl"
2. Point "dblPtr" to "dbl" (Assign the address of "dbl" to be stored in "dblPtr")
3. Print the value of "dbl"
4. Print the value of "dblPtr"
5. Print the address of "dbl"
6. Print the address of "dblPtr"
7. Print the sizeof "dbl"
8. Print the sizeof "dblPtr"

We can actually store the address of a pointer in a variable. This requires a variable is known as a pointer to a pointer (e.g. double **) and is the basis for two dimensional arrays. But, we'll leave that discussion for the future.

Pointer values can be changed

We can change a pointer to point to a new variable if we choose. Add a second double variable named “dbl2” to “pointers1-2.cpp”. Add the following instructions, at the end of the file, along with appropriate comments.

1. Point “dblPtr” to “dbl2”
2. Print the value of “dbl2”
3. Print the value of “dblPtr”
4. Print the address of “dbl2”
5. Print the address of “dblPtr”

Compile and run your code. Then, submit your code below.

Pointers can be used with the mathematical operators

We can add and subtract values to and from pointers. This is usually done in conjunction with memory allocated contiguously (i.e. an array), but we are just going to observe how the values of our pointers change for now.

Remove the previous file from your project and add a new file named “pointers1-3.cpp” to the project and type in the following code. Make sure you write the proper comments for the file using “pointers1-3.cpp” as the name of the file and “Pointer Arithmetic” as the description for the file. Also, add comments within the code to explain what is going on.

```
#include <iostream>

using namespace std;

int main() {
    int x, *intPtr = &x;
    char ch, *chPtr = &ch;
    double dbl, *dblPtr = &dbl;

    cout << intPtr << " is the initial value of intPtr" << endl;
    cout << (int*)chPtr << " is the initial value of chPtr" << endl;
    cout << dblPtr << " is the initial value of dblPtr" << endl;

    intPtr++;
    chPtr++;
    dblPtr++;

    cout << intPtr << " is the new value of intPtr" << endl;
    cout << (int*)chPtr << " is the new value of chPtr" << endl;
    cout << dblPtr << " is the new value of dblPtr" << endl;
```

```
    return 0;
}
```

Compile and run your code. Then, submit your code below.

Answer the following question in your .h file. Restate the question immediately before your answer please. 1. What are differences between the original value and the incremented value? 2. Is this what you expected? 3. Provide an explanation of the results.

Pointer introduction to indirection

We can use pointers to view and even change that value to which it points. The indirection operator (*) is used to gain access to the value stored at that address.

Remove the previous file from the project and add a new .cpp file named “pointers1-4.cpp” to the project. Make sure to add all appropriate comments using “Pointer Introduction to indirection” as the description. Also, add comments within the code to explain what is going on.

```
#include <iostream>

using namespace std;

int main() {
  int x, *intPtr;
  char ch, *charPtr;

  x = 10;
  intPtr = &x;

  ch = 'a';
  charPtr = &ch;

  cout << "The value of x is " << x << endl;
  cout << "The value of x using intPtr is " << *intPtr << endl;

  cout << "The value of ch is " << ch << endl;
  cout << "The value of ch using charPtr is " << *charPtr << endl;

  return 0;
}
```


Indirection can be used to change values

Add the following instructions to “pointers1-4.cpp” to the end of the program.

1. Using only the indirection operator and “intPtr”, change the value of “x” to 0.

2. Using only the indirection operator and “charPtr”, change the value of “ch” to ‘Z’
3. Print the value of x to the screen (make sure to use x not intPtr).
4. Print the value of ch to the screen (make sure to use ch not charPtr)

Compile and run your code. Then, submit your code below.

Pointers passed to functions

If pointers are variables and pointers can be used to change values, then we can pass addresses of variables to functions and make changes to the variables in the functions. In fact, this is alternative way to passing by reference.

Remove the previous file from the project and add a new .cpp file named “pointers1-5.cpp” to the project. Make sure to add all the appropriate comments using “Pointers as Parameters” as the description. Add comments within the code to explain what is going on. And, provide the proper comments for your function. Here is an example of proper function commenting.

```
#include <iostream>
#include <cmath>

using namespace std;

/**
-  changeIt
-
-  This function changes the value of the parameter passed to it using
-    pointers.
-
-  Parameters:
-    n: an address of variable declared locally in the calling function
-
-  Output:
-    return: none
-    parameters: n - the changed value at the address passed
-    stream: none
*/
void changeIt( int *n );

int main() {
   int x, y, z;

   x = y = z;

   cout << "x's value was " << x << " before a call to changeIt()" <<
endl;
```

```
   changeIt( &x );
   cout << "x's value is " << x << " after a call to changeIt()" <<
endl;

   cout << "y's value was " << y << " before a call to changeIt()" <<
endl;
   changeIt( &y );
   cout << "y's value is " << y << " after a call to changeIt()" <<
endl;

   cout << "z's value was " << z << " before a call to changeIt()" <<
endl;
   changeIt( &z );
   cout << "z's value is " << z << " after a call to changeIt()" <<
endl;

   return 0;
}
```


Write a function named “changeIt”. Have “changeIt” assign a random value to the value at the address passed. The random value should be 0 or greater but less than 100.

Compile and run your code. Then, submit your code below.

Answer the following question document. 4. Explain the difference between *, &, and just the variable’s name. Then, submit your .h file below.

LAB ACTIVITY 10.14.1: Lab 1 - Pointer Introduction 0 / 1

Submission Instructions

Downloadable files

`lab1-pointers1-3.cpp` , `lab1-pointers1-1.cpp` , `lab1-pointers1-5.cpp` , `lab1-pointers1-4.cpp` , and `lab1-answersToQuestions.h` **Download**

Compile command

```
g++ lab1-pointers1-3.cpp lab1-
pointers1-1.cpp lab1-pointers1-2.cpp
lab1-pointers1-5.cpp lab1-pointers1-
```

We will use this command to compile your code

```
4.cpp lab1-pointers1.cpp -Wall -o
a.out
```

Upload your files below by dragging and dropping into the area or choosing a file on your hard drive.

lab1-pointers1-3.cpp

Drag file here
or
Choose on hard drive.

lab1-pointers1-1.cpp

Drag file here
or
Choose on hard drive.

lab1-pointers1-2.cpp

Drag file here
or
Choose on hard drive.

lab1-pointers1-5.cpp

Drag file here
or
Choose on hard drive.

lab1-pointers1-4.cpp

Drag file here
or
Choose on hard drive.

lab1-pointers1.cpp

Drag file here
or
Choose on hard drive.

lab1-answersToQuestions.h

Drag file here
or
Choose on hard drive.

Submit for grading

Signature of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```


Latest submission

No submissions yet

10.15 Lab 2 - Dynamic Memory

For this lab, we will not use zyBooks' testing functionality again. All testing will come from your local computer.

Last lab we gained some basic experience with pointers. We observed that pointers store addresses and allow us to access the values stored at those addresses. We gained experience assigning addresses to pointers and even moving pointers around using pointer arithmetic. Lastly, we learned that there are strong similarities between pointers and arrays.

This week we will explore the relationship between pointers and arrays in more depth. We will also explore how to set up pointers with memory from the heap (dynamic memory).

First, we need to discuss the requirements for this lab. You will need to use the .h file to record your answers to the questions. To get full credit, you will need to start the .h file off by including the following information in the top left corner of the first page.

Name

CSI 1440 Section X (where X is your section number)

Lab 2

You will be required to provide answers to questions throughout the lab.

Pointers and arrays

Pointers and arrays are very similar, but they are different. Type in and execute the code in the following sections to see differences between pointers and arrays.

Step 1:

```
#include <iostream>

using namespace std;

int main() {
  char myArray[] = "Hello World!";
  char *myPtr = "Hello World!";

  cout << "This is from the array: " << myArray << endl;
  cout << "This is from the pointer: " << myPtr << endl;

  return 0;
}
```


Notice that we may get a warning from this code, but it does seem from code like this that pointers and arrays are the same.

Step 2:

```
#include <iostream>

using namespace std;

int main() {
  char myArray[] = "Hello World!";
  char *myPtr = "Hello World!";

  cout << "This is from the array: " << myArray << endl;
  cout << "This is from the pointer: " << myPtr << endl;
  cout << endl;

  cout << "Trying to move the pointer..." << endl;
  myPtr += 6;
  cout << "This is from the pointer: " << myPtr << endl;

  return 0;
}
```


Compile and execute the code. Answer the following question within your .h file. Make sure to type the question immediately before your answer.

1. What would be printed if we just incremented the pointer (myPtr++)?

Step 3:

```
#include <iostream>

using namespace std;

int main() {
  char myArray[] = "Hello World!";
  char *myPtr = "Hello World!";

  cout << "This is from the array: " << myArray << endl;
  cout << "This is from the pointer: " << myPtr << endl;
  cout << endl;

  cout << "Trying to move the array ..." << endl;
  myArray += 6;
  cout << "This is from the array: " << myArray << endl;
  return 0;
}
```

Compile the code. C++ will not allow you to change the address stored by the array. It is effectively a constant value when you declare an array.

Submit a version of this code using “pointers2-1.cpp” as the filename. Provide the answer to the following question in your .h file.

2. How does “myArray += 6” differ from what we talked about in class - “*(myArray + 6)” - which we said was OK?

Step 4:

```
#include <iostream>

using namespace std;

int main() {
  char myArray[] = "Hello World!";
  char *myPtr = "Hello World!";

  cout << "This is from the array: " << myArray << endl;
  cout << "This is from the pointer: " << myPtr << endl;

  cout << endl << "Let's try to change the array..." << endl;
  myArray[1] = 'o';
  myArray[2] = 'w';
  myArray[3] = 'd';
  myArray[4] = 'y';

  cout << "This is from the array: " << myArray << endl;

  return 0;
}
```

Compile and run. So, we introduced a Texas feel to the “Hello World!” testing of arrays and pointers. Show the changes requested in the following statement within your .h file.

3. Replace the array notation statements with pointer notation statements for the assignment of ‘o’, ‘w’, ‘d’, and ‘y’.

Step 5:

Let’s try it with the pointer now.

```
#include <iostream>

using namespace std;

int main() {
```

```
  char myArray[] = "Hello World!";
  char *myPtr = "Hello World!";

  cout << "This is from the array: " << myArray << endl;
  cout << "This is from the pointer: " << myPtr << endl;

  cout << endl << "Let's try to change the pointer..." << endl;
  myPtr[1] = 'o';
  myPtr[2] = 'w';
  myPtr[3] = 'd';
  myPtr[4] = 'y';

  cout << "This is from the pointer: " << myPtr << endl;

  return 0;
}
```


Why did this happen? It turns out that the pointer has been storing the address of the string literal "Hello World!" which is a constant. Our attempt to change "Hello" to "Howdy" is identical to writing something like this: "5 = 4" which makes no sense at all. In this case we are actually telling the computer to make 'e' = 'o', 'l' = 'w', 'l' = 'd', and 'o' = 'y'.

Answer the following question in your .h file.

4. Could this code be fixed by using pointer notation instead of array notation? Why/Why not?

How can we change this behavior? Well, we could use an array to create the memory. Assign the pointer to that memory, and move the pointer around. We would try this scenario, but we already have in the previous Lab. So, let's try something a little different. Let's try to get the memory from within a function.

Dynamic Memory

Step 1:

```
#include <iostream>

using namespace std;

char* giveMemory( ) {
  char myArray[] = "Hello World!";
  return myArray;
}

int main() {
  char *myPtr;
```

```
  myPtr = giveMemory();
  cout << myPtr << endl;

  myPtr[1] = 'o';
  myPtr[2] = 'w';
  myPtr[3] = 'd';
  myPtr[4] = 'y';

  cout << endl << myPtr << endl;

  return 0;
}
```


Submit a version of this code using “pointers2-2.cpp” as the filename .

What went wrong? It turns out that the memory associated with an array is allocated on the run-time stack. I remind you of the discussion we had on functions last semester. Each time a function is called. The function’s stack frame is placed on the run-time stack. The function’s stack frame contains all locally declared variables’ memory. An array’s memory is treated similarly to that of a regular variable - it is located within the stack - which prevents us from returning a reference to that memory from the function.

How do we solve this problem? With dynamic memory. Dynamic memory is allocated on the Heap. Yes, that structure we didn’t talk about much last semester. The memory we get from the Heap will be available to us until we tell the computer we are done with it.

Step 2:

```
#include <iostream>
#include <cstring>

using namespace std;

char* giveMemory( ) {
  char *myArray = new char[13];
  strcpy( myArray, "Hello World!");
  return myArray;
}

int main() {
  char *myPtr;

  myPtr = giveMemory();

  cout << myPtr << endl;
```

```
  myPtr[1] = 'o';
  myPtr[2] = 'w';
  myPtr[3] = 'd';
  myPtr[4] = 'y';

  cout << endl << myPtr << endl;

  delete [] myPtr;

  return 0;
}
```


Notice that we had to ask for 13 characters using the new operator. We also had to hand back all the memory we ask for with the delete operator.

Answer the following question within your .h file.

5. Why did we not give the memory back to the computer, using the delete operator, within the function?

Just for fun, let's go back to the first series of programs and use dynamic memory in them.

```
#include <iostream>
#include <cstring>

using namespace std;

int main() {
  char myArray[] = "Hello World!";
  char *myPtr = new char[13];

  strcpy(myPtr, "Hello World!");

  cout << "This is from the array: " << myArray << endl;
  cout << "This is from the pointer: " << myPtr << endl;

  cout << endl << "Let's try to change the array..." << endl;
  myArray[1] = 'o';
  myArray[2] = 'w';
  myArray[3] = 'd';
  myArray[4] = 'y';

  cout << "This is from the array: " << myArray << endl;

  cout << endl << "Let's try to change the pointer..." << endl;
  myPtr[1] = 'o';
```

```
  myPtr[2] = 'w';
  myPtr[3] = 'd';
  myPtr[4] = 'y';

  cout << "This is from the pointer: " << myPtr << endl;

  delete [] myPtr;

  return 0;
}
```


Submit a version of this code using “pointers2-3.cpp” as the filename.

Answer the following question in your .h file.

6. Why do we not need to use the indirection operator when printing the contents of myArray and myPtr? Could we print the contents of a differently typed (int, double, etc..) array?

Creating and Accessing Dynamic Objects

We just demonstrated how to create, use, and free dynamic memory for primitive data types. Now we need to see how we create, use, and free dynamic objects. Dynamically created objects can be used identically to their primitive cousins, but we have the added complexity of the public interface. The added complexity lead to the inclusion of an additional operator. The additional operator is called the “structure pointer operator”, or “arrow operator”. Here is a short example for you to type in.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string *str = new string("Hello World!");

  // using the indirection operator to access the public interface of
string
  cout << (*str).length() << endl;

  // using the structure pointer operator to access the public
interface of string
  cout << str->length() << endl;

  delete str;

  return 0;
}
```

Submit a version of this code using “pointers2-4.cpp” as the filename.

Answer the following question in your .h file.

7. Write two additional lines of code to print out the contents of str as a c-string representation using c_str() with both the indirection operator and the structure pointer operator.

LAB ACTIVITY | 10.15.1: Lab 2 - Dynamic Memory 0 / 1

Submission Instructions

Compile command

```
g++ lab2-pointers2-2.cpp lab2-pointers2-1.cpp lab2-pointers2-4.cpp lab2-pointers2-3.cpp -Wall -o a.out
```

We will use this command to compile your code

Upload your files below by dragging and dropping into the area or choosing a file on your hard drive.

lab2-pointers2-2.cpp
Drag file here
or
Choose on hard drive.

lab2-pointers2-1.cpp
Drag file here
or
Choose on hard drive.

lab2-pointers2-4.cpp
Drag file here
or
Choose on hard drive.

lab2-pointers2-3.cpp
Drag file here
or
Choose on hard drive.

lab2-answersToQuestions.h
Drag file here
or
Choose on hard drive.

Submit for grading

Signature of your work What is this?

```
History of your effort will appear here once you begin working on this zyLab.
```

Latest submission

No submissions yet

10.18 Lab 3 - C-string functions

CSI 1440 Lab 3 “C-string functions”

We have been studying and testing pointers, dynamic memory, and classes with pointers and dynamic memory. This lab we will be using pointers and dynamic memory with c-strings. More specifically we will be implementing several c-string functions that we will be discussing in class as well.

Beginner c-string functions

The first set of functions we are going to write, I consider to be the easiest. We will implement them two different ways. I want each of you to concentrate not on memorizing the code, but instead, concentrating on the algorithm. You will encounter several problems that are solved with similar algorithms in the future. It will serve each of you best if you can first understand the algorithm presented, then create an efficient solution from scratch. Most of these functions are made available in the cstring library, just include cstring to gain access. Of course, we will be writing our own versions. So, there is no need to include cstring for this lab.

strlen():

strlen() counts the number of characters in a cstring. The algorithm works by looking at each character in the cstring, counting as it goes. It stops looking at characters when it encounters a null terminator ‘\0’.

Here is my implementation of the strlen() function. You will need to write and test my solution and your own solution to get full credit for the lab.

```
int profStrLen( char *str ) {
  char *endOfStr = str;
  while( *endOfStr != '\0' ) {
     endOfStr++;
  }
  return (int)(endOfStr - str);
}
```


Make sure you understand how my implementation is working before creating your own. Then, you can write your own implementation. For your implementation, do not use pointer arithmetic to solve it - use array notation and a counter instead. Here is the prototype for your function.

```
int stuStrLen( char * );
```

strcpy():

strcpy() copies the contents of the second char* into the memory of the first char*. The algorithm works by looking at each character in the second char* and copies that into the first char*. Because we are dealing with cstrings, the copying process will stop when the '\0' is reached in the second char*.

Here's my implementation. Again, you will need to write and test my solution and your own to get full credit.

```
char *profStrCpy( char *str1, char *str2 ) {
  char *dest = str1;
  while( *str2 != '\0' ) {
    *str1 = *str2;
    str1++;
    str2++;
  }
  *str1 = '\0';
  return dest;
}
```

Make sure you understand how my implementation is working before creating your own. Then, you can write your own implementation. For your implementation, do not use pointer arithmetic to solve it - use array notation and a counter instead. Here is the prototype for your function.

```
char *stuStrCpy( char *, char * );
```

strcat():

strcat() copies the contents of the second char* into the memory of the first char* like strcpy(), but it copies the values in the second char* to the end of the first char*. The algorithm works by looking at each character in the first char* until the '\0' is found. Then, it looks at each character in the second char* and copies that into the first char*'s current location. As it does this, it increments the position in each char* until the '\0' is found in the second char*.

Here's my implementation. Again, you will need to write and test my solution and your own to get full credit.


```
char *profStrCat( char *str1, char *str2 ) {
  char *dest = str1;
  while( *str1 != '\0' ) {
    str1++;
  }
  while( *str2 != '\0' ) {
```

```
    *str1 = *str2;
    str1++;
    str2++;
  }
  *str1 = '\0';
  return dest;
}
```


Make sure you understand how my implementation is working before creating your own. Then, you can write your own implementation. For your implementation, do not use pointer arithmetic to solve it - use array notation and a counter instead. Here is the prototype for your function.

```
char *stuStrCat( char *, char * );
```

Intermediate cstring function

Now that you have worked through many of the cstring functions. I want each of you to implement the following functions on your own. I will provide an overview of the algorithm, but I will not provide you with an implementation from me. You will need to write and test this function to get full credit.

strcmp():

strcmp() compares strings lexicographically. It returns a negative value if the first string occurs lexicographically before the second string. It returns a positive value if the first string occurs lexicographically after the second string. And, it returns zero if the two strings are equivalent. Make sure you call attention to the return of strcmp(). The return of zero instead of a non-zero is a common source for errors involved with cstring processing.

The algorithm for strcmp() works by subtracting each character as it iterates through each cstring stopping when the result of the subtraction is nonzero or a '\0' is found in one of the strings. We will return the difference between the two characters.

You will need to write and test your solution to get full credit. Here is your prototype.

```
int stuStrCmp( char *, char * );
```

Everybody likes a mystery

Here is a function of code that is labeled mystery. Type in the code and describe in the comments what the function does algorithmically and tell me what you think the function is named.


```
int mystery( char *str ) {
  char *str2 = str;
  long pos;
  int result = 0;

  while( *str2 != '\0' ) {
    str2++;
```

```
  }
  pos = str2 - str - 1;

  for( int count = 1; pos >= 0; pos --, count *= 10 ) {
    if( str[pos] == '-') {
      result *= -1;
    } else {
      result += (str[pos] - '0') * count;
    }
  }
  return result;
}
```


LAB ACTIVITY | 10.18.1: Lab 3 - C-string functions | 0 / 1

main.cpp — Load default template...

```
#include <iostream>

using namespace std;

int profStrLen( char * );
int stuStrLen( char * );
char *profStrCpy( char *, char * );
char *stuStrCpy( char *, char * );
char *profStrCat( char *, char * );
char *stuStrCat( char *, char * );
int stuStrCmp( char *, char * );
int mystery( char * );

int main() {
   // type all testing code here
   return 0;
}
```

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) → Outp

Program output displayed here

Signature of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```


©zyBooks 04/25/21 07:06 488201
xiang zhao
BAYLORCSI14301440Spring2021