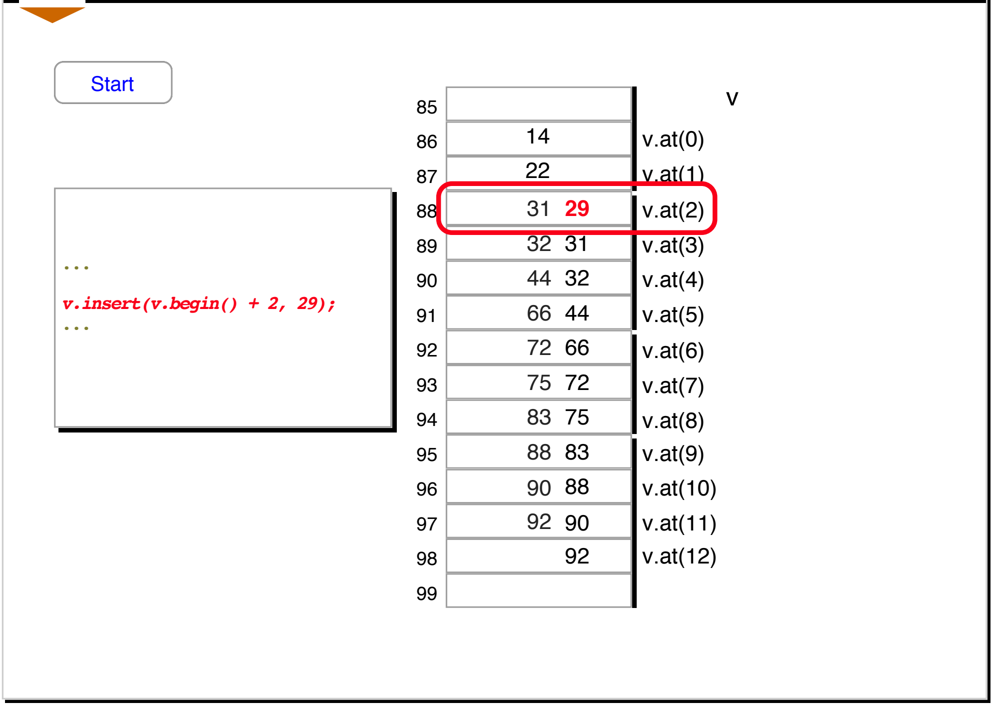
Chapter 8 - Pointers

Section 8.1 - Why pointers: A list example

A challenging and yet powerful programming construct is something called a *pointer*. This section describes one of many situations where pointers are useful.

A vector (or array) stores a list of items in contiguous memory locations. Storing in contiguous locations enables immediate access to any element of vector v by using `v.at(i)` (or `v[i]`), because the compiler just adds i to the starting address of v to access the element at index 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 processor instructions. For vectors with thousands of elements, a single call to insert() or erase() can require thousands of instructions, so a program with many inserts or erases on large vectors may run very slowly, what we call the ***vector insert/erase performance problem***.

P Participation Activity 8.1.1: Vector insert performance problem.



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

The following program demonstrates. The user inputs a vector size, and a number of operations (numOps) to perform. The program then resizes the vector, writes a value to each element, does numOps push_backs, does numOps inserts, and does numOps erases. The `<< flush` forces cout to ***flush*** any characters in its buffer to the screen before doing each task, otherwise the characters may be held in the buffer until after a later task completes. Running the program for vectorSize of 100000 and numOps 40000 shows that the writes and push_backs execute fast, but the inserts and erases are noticeably slow.

Figure 8.1.1: Program illustrating that vector inserts and erases can be slow.

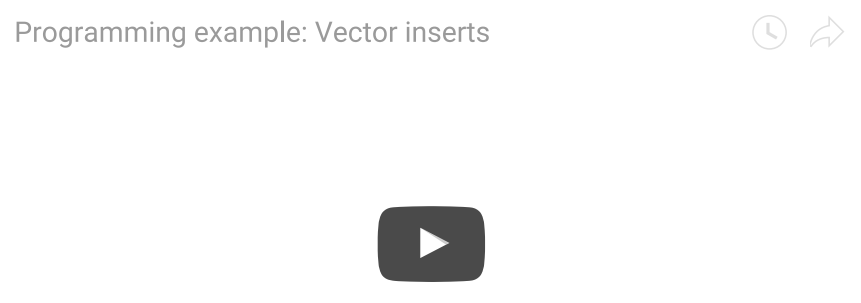
```
#include <iostream>
#include <vector>
using namespace std;

int main() {
   vector<int> tempValues; // Dummy vector to demo vector ops
   int vectorSize = 0;     // User defined size
   int numOps = 0;         // Number of operations to perform
   int i = 0;              // Loop index

   cout << "Enter initial vector size: ";
   cin >> vectorSize;

   cout << "Enter number of operations: ";
   cin >> numOps;

   cout << "  Resizing vector..." << flush;

   tempValues.resize(vectorSize);

   cout << "done." << endl;
   cout << "  Writing to each element..." << flush;

   for (i = 0; i < vectorSize; ++i) {
      tempValues.at(i) = 777; // Any value
   }

   cout << "done." << endl;
   cout << "  Doing " << numOps << " pushbacks..." << flush;

   for (i = 0; i < numOps; ++i) {
      tempValues.push_back(888); // Any value
   }

   cout << "done." << endl;
   cout << "  Doing " << numOps << " inserts..." << flush;

   for (i = 0; i < numOps; ++i) {
      tempValues.insert(tempValues.begin() + 0, 444);
   }
   cout << "done." << endl;
   cout << "  Doing " << numOps << " erases..." << flush;

   for (i = 0; i < numOps; ++i) {
      tempValues.erase(tempValues.begin() + 0);
   }

   cout << "done." << endl;

   return 0;
}
```

```
Enter initial vector size: 100000
Enter number of operations: 40000
  Resizing vector...done.            (fast)
  Writing to each element...done.    (fast)
  Doing 40000 pushbacks...done.      (fast)
  Doing 40000 inserts...done.        (SLOW)
  Doing 40000 erases...done.         (SLOW)
```

The push_backs are fast because they do not involve any shifting of elements, whereas each insert requires 100,000 elements to be shifted, one at a time. 40,000 inserts thus requires 4,000,000,000 shifts.

The video shows the program running for different vector sizes and number of operations; notice that for large values, the resize, writes, and push_backs all run quickly, but the inserts and erases take a noticeably long time.

Video 8.1.1: Vector inserts.



P Participation Activity

8.1.2: Vector insert/erase problem.

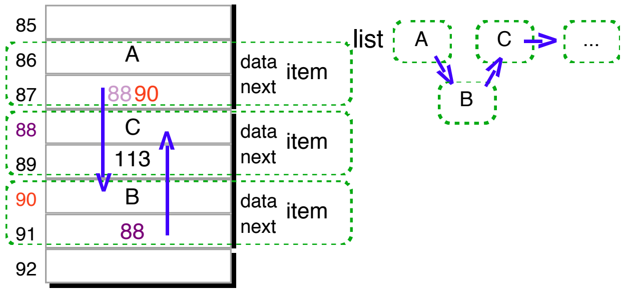
For each operation, how many elements must be shifted? Assume no new memory needs to be allocated. Questions are for vectors, but apply to arrays too.

#	Question	Your answer
1	Append an item to the end of a 999-element vector (e.g., using push_back()).	
2	Insert an item at the front of a 999-element vector.	
3	Delete an item from the end of a 999-element vector.	
4	Delete an item from the front of a 999-element vector.	

One way to make inserts or erases faster is to use a different approach for storing a list of items. The approach does not use contiguous memory locations. Instead, each item contains a "pointer" to the next item's location in memory, as well as the data being stored. 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.

Participation Activity

8.1.3: A list avoids the shifting problem.



Add new item B at location 90.
Change item A to point to 90.
Set item B to point to 88.

New list is (A, B, C, ...) -- no shifting of items after C

The initial list contains an item with data A followed by an item with C. Inserting item B did not require C to be shifted.

A ***linked list*** is a list wherein each item contains not just data but also a pointer—a *link*—to the next item in the list. Comparing vectors and linked lists:

- *Vector*: Stores items in contiguous memory locations. Supports quick access to i'th element via `v.at(i)`, but may be slow for inserts or deletes on large lists due to necessary shifting of elements.
- *Linked list*: Stores each item anywhere in memory, with each item pointing to the next item in the list. Supports fast inserts or deletes, but access to i'th element may be slow as the list must be traversed from the first item to the i'th item. Also uses more memory due to storing a link for each item.

A vector/array 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.

Participation Activity

8.1.4: Linked list inserts/deletes using pointers.

#	Question	Your answer
1	Appending an item at the end of a 999-item linked list requires how many items to be shifted?	
2	Inserting a new item between the 10th and 11th items of a 999-item linked list will require a few pointer changes. In addition, how many shifts will be required?	
3	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.	

Exploring further:

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

Section 8.2 - Pointer basics

A ***pointer*** is a variable that contains a memory address, rather than containing data like most variables introduced earlier. The following program introduces pointers via example:

Figure 8.2.1: Introducing pointers via a simple example.

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

int main() {
   int usrInt = 0; // User defined int value
   int* myPtr = 0; // Pointer to the user defined int value

   // Prompt user for input
   cout << "Enter any number: ";
   cin >> usrInt;

   // Output int value and address
   cout << "We wrote your number into variable usrInt." << endl;
   cout << "The content of usrInt is: " << usrInt << "." << endl;
   cout << "usrInt's memory address is: " << &usrInt << "." << endl;
   cout << endl << "We can store that address into pointer variable myPtr."
        << endl;

   // Grab address storing user value
   myPtr = &usrInt;

   // Output pointer value and value at pointer address
   cout << "The content of myPtr is: " << myPtr << "." << endl;
   cout << "The content of what myPtr points to is: "
   << *myPtr << "." << endl;

   return 0;
}
```

```
Enter any number: 555
We wrote your number into variable usrInt.
The content of usrInt is: 555.
usrInt's memory address is: 0x7fff5fbff888.

We can store that address into pointer variable myPtr.
The content of myPtr is: 0x7fff5fbff888.
The content of what myPtr points to is: 555.
```

The example demonstrates key aspects of working with pointers:

- *Appending * after a data type* in a variable definition defines a pointer variable, as in `int* myPtr`. One might imagine that the programming language would have a type like *address* in addition to types like int, char, etc., but instead the language requires each pointer variable to indicate the type of data to which the address points. So valid pointer variable definitions are `int* myPtr1`, `char* myPtr2`, `double* myPtr3`, and even `Seat* myPtr4;` (where Seat is a class type); all such variables will contain memory addresses.
- *Prepending & to any variable's name gets the variable's address*. & is the reference operator that returns a pointer to a variable using the following form:

Construct 8.2.1: Reference operator.

```
&variableName
```

- *Prepending * to a pointer variable's name in an expression gets the data to which the variable points*, as in `*myPtr1`, an act known as ***dereferencing*** a pointer variable. * is the dereference operator that allows the program to access the value pointed to by the pointer using the form:

Construct 8.2.2: Dereference operator.

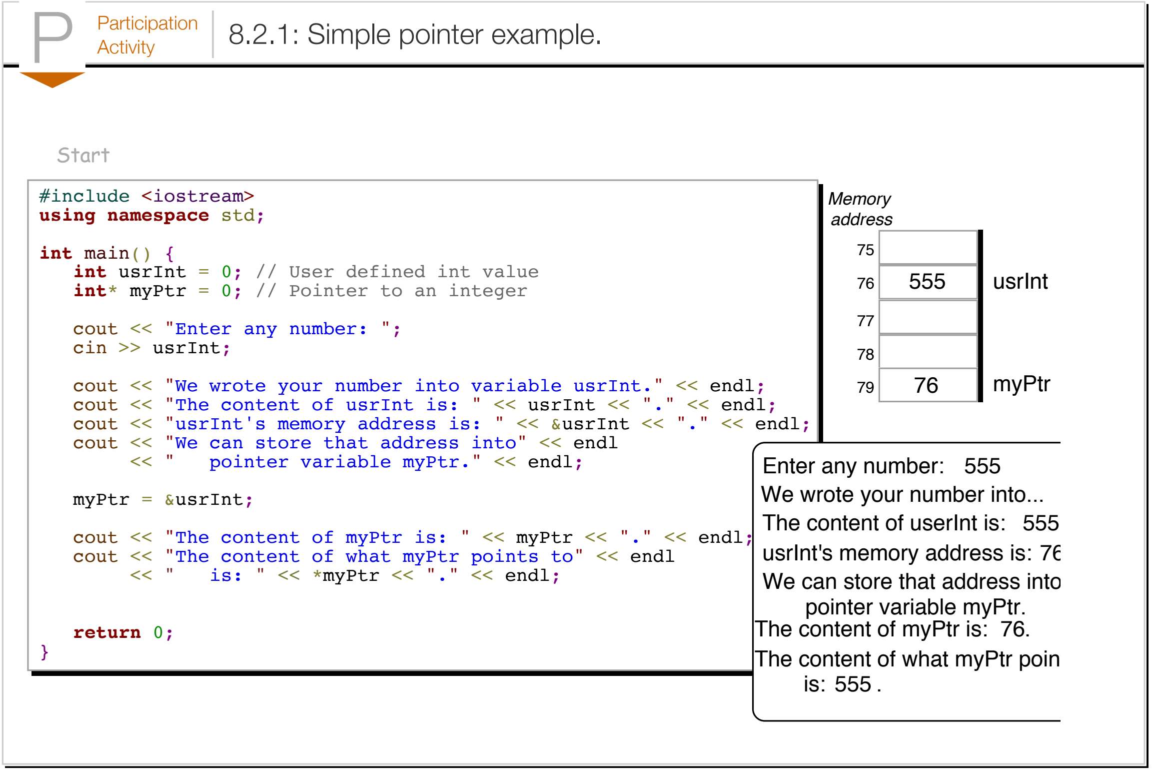
```
*variableName
```

Observe the above program's output. For int variable usrInt, the statement `cout << &usrInt;` prints usrInt's memory address, which is the large number 0x7fff5fbff888 in contrast to short memory addresses like 96 that have appeared in earlier animations. That large number is in hexadecimal or base 16 number, which you need not concern yourself with as you will not normally print or ever have to look at such memory addresses; the memory address is printed here just for illustration. The statement `myPtr = &usrInt;` will thus set myPtr's contents to that large address. The statement `cout << myPtr;` will print myPtr's contents, which is that large address. The statement `cout << *myPtr;` will instead go to that address and then print that address' contents.

The * (asterisk) symbol is used in two ways related to pointers. One is to indicate that a variable is a pointer type, as in `int* myPtr`. The other is to dereference a pointer variable, as in `cout << *myPtr`. Don't be confused by those two different uses; they have different meanings, both related to pointers.

The pointer was initialized to 0. Because 0 is not a valid memory address, it can be used to indicate that the pointer variable points to nothing. The pointer is said to be ***null***. Note_null

The following animation illustrates pointers.



Participation Activity

8.2.1: Simple pointer example.

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

int main() {
   int usrInt = 0; // User defined int value
   int* myPtr = 0; // Pointer to an integer

   cout << "Enter any number: ";
   cin >> usrInt;

   cout << "We wrote your number into variable usrInt." << endl;
   cout << "The content of usrInt is: " << usrInt << "." << endl;
   cout << "usrInt's memory address is: " << &usrInt << "." << endl;
   cout << "We can store that address into" << endl
        << "   pointer variable myPtr." << endl;

   myPtr = &usrInt;

   cout << "The content of myPtr is: " << myPtr << "." << endl;
   cout << "The content of what myPtr points to" << endl
        << "   is: " << *myPtr << "." << endl;


   return 0;
}
```

```
Enter any number:   555
We wrote your number into...
The content of userInt is:   555
usrInt's memory address is: 76
We can store that address into
      pointer variable myPtr.
The content of myPtr is:  76.
The content of what myPtr poin
      is:  555 .
```

The * in a pointer variable definition has some syntactical options. We wrote `int* myPtr`. However, also allowed is `int *myPtr`. Many programmers find the former option that groups the int and * more intuitive, suggesting myPtr is of type "integer pointer". On the other hand, note that `int* myPtr1, myPtr2` does *not* define two pointers, but rather defines pointer variable myPtr1, and int variable myPtr2. For this reason, some programmers prefer the option that groups the * with the variable name, as in `int *myPtr1, *myPtr2`. Our advice: To reduce errors, it may be good practice to only define one pointer per line, using the int* option.

P Participation Activity

8.2.2: Using pointers.

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

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

```

#include <iostream>
using namespace std;

int main() {
   double vehicleMpg = 0.0;
   double* valPtr = 0;

   valPtr = &vehicleMpg;

   *valPtr = 29.6; // Assigns the number to the variable
                   // POINTED TO by valPtr.

  // vehicleMpg = 40;   // Uncomment this later

   cout << "Vehicle MPG = " << vehicleMpg << endl;
   cout << "Vehicle MPG = " << *valPtr << endl;
   return 0;
}
```

Run

P Participation Activity

8.2.3: Pointer basics.

Assume variable int numStudents = 12 is at memory address 99, and variable int* myPtr is at address 44. Answer "error" where appropriate.

#	Question	Your answer
1	What does `cout << numStudents` output?	
2	What does `cout << &numStudents` output?	
3	What does `cout << *numStudents` output?	
4	After `myPtr = &numStudents`, what does `cout << *myPtr` output?	

C Challenge Activity

8.2.1: Printing with pointers.

Assign numItems' address to numItemsPtr, then print the shown text followed by the value to which numIt
newline.

```
Items: 99
```

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

int main() {
   int* numItemsPtr = 0;
   int numItems = 99;

   /* Your solution goes here  */

   return 0;
}
```

Run

(*Note_null) Note to instructors: We initialize the pointer to 0, avoiding use of NULL per C++ creator Bjarne Stroustrup's advice: "In C, it has been popular to define a macro *NULL* to represent the zero pointer. Because of C++'s tighter type checking, the use of plain *0*, rather than any suggested *NULL* macro, leads to fewer problems." (From "The C++ Programming Language," Bjarne Stroustrup, AT&T, 1997). Actually, the nullptr keyword (see Wikipedia: C++11) in newer versions of the C++ standard would be even better, but is not yet supported by many C++ compilers.

Section 8.3 - Operators: new, delete, and ->

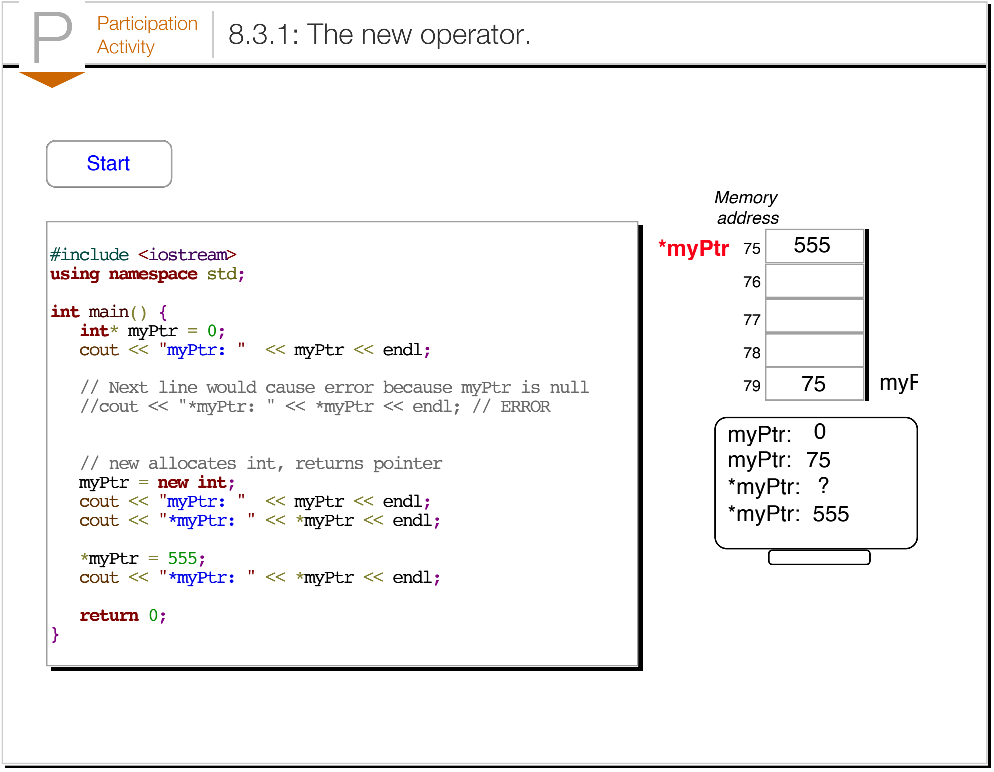
new: allocating memory

Sometimes memory should be allocated while a program is running and should persist independently of any particular function. The ***new*** operator allocates memory for the given type and returns a pointer (i.e., the address) to that allocated memory.

Construct 8.3.1: The new operator.

```
pointerVariable = new type;
```

The following animation illustrates using new to allocate memory of an int type. The int type is used for introduction; new is more commonly used to allocate memory for a class type.



8.3.1: The new operator.

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

int main() {
   int* myPtr = 0;
   cout << "myPtr: "  << myPtr << endl;

   // Next line would cause error because myPtr is null
   //cout << "*myPtr: " << *myPtr << endl; // ERROR


   // new allocates int, returns pointer
   myPtr = new int;
   cout << "myPtr: "  << myPtr << endl;
   cout << "*myPtr: " << *myPtr << endl;

   *myPtr = 555;
   cout << "*myPtr: " << *myPtr << endl;

   return 0;
}
```

```
myPtr:   0
myPtr:  75
*myPtr:  ?
*myPtr:  555
```

The new operator returns 0 if the operator failed to allocate memory. Such failure could happen if a program has used up all memory available to the program.

The new operator is commonly used with class types, as in `new myItem;` where myItem is a class name. The new operator allocates the appropriate block of memory for the class, calls the constructor for the class, and returns a pointer to that block. Arguments may be provided after the class name to call a non-default constructor.

Figure 8.3.1: Using the new operator with a class type.

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

class myItem {
public:
   void PrintNums();
   myItem(int initVa1 = -1, int initVal2 = -1);
private:
   int num1;
   int num2;
};

myItem::myItem(int initVal1, int initVal2) {
   num1 = initVal1;
   num2 = initVal2;
   return;
}

void myItem::PrintNums() {
   cout << "num1: " << num1 << endl;
   cout << "num2: " << num2 << endl;
   return;
}

int main() {
   myItem* myItemPtr1 = 0;
   myItem* myItemPtr2 = 0;

   myItemPtr1 = new myItem;
   (*myItemPtr1).PrintNums();
   cout << endl;

   myItemPtr2 = new myItem(8, 9);
   (*myItemPtr2).PrintNums();

   return 0;
}
```

```
num1: -1
num2: -1

num1: 8
num2: 9
```

->: member access operator

Accessing a class's member functions by first dereferencing a pointer, as in `(*myItemPtr1).PrintNums()`, is so common that the language includes a second ***member access operator***, in particular the **->** operator that allows an alternative to `(*a).b`:

Construct 8.3.2: Member access operator.

```
a->b    // Equivalent to (*a).b
```

Thus the above program could have used: `myItemPtr1->PrintNums();`.

delete: deallocating memory

The ***delete*** operator does the opposite of the new operator. The statement `delete pointerVariable;` deallocates a memory block pointed to by pointerVariable, which must have been previously allocated by new.

Construct 8.3.3: Delete operator.

```
delete pointerVariable;
```

After the delete, the program should not attempt to dereference pointerVariable, as 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—if that memory had since been allocated to another variable, that variable's value could mysteriously change. Calling delete with a pointer that wasn't *previously* set by the new operator is also an error.

The following example illustrates a common use of new and delete in conjunction with a vector storing items of a class type. The new operator is used to allocate memory for a new item, which is then added to the vector. The delete operator deletes the memory for the item, before removing the item from the vector. The example implements a simple inventory management system in which items can be added or removed from an inventory list.

Figure 8.3.2: Inventory management with new and delete operators.

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

class myItem {
public:
   void PrintItem();
   myItem(string initName = "", int initQty = 0);
private:
   string itemName;  // Name of item
   int itemQuantity; // Number of items available
};

// myItem Constructor
myItem::myItem(string initName, int initQty) {
   itemName = initName;
   itemQuantity = initQty;
   return;
}

// myItem function to print name/qty attributes
void myItem::PrintItem() {
   cout << "name: " << this->itemName << ", " << "quantity: "
        << this->itemQuantity << endl;
   return;
}

// Displays all items currently stored in vector itemsInventory
void PrintAllItems(vector<myItem*> itemsInventory) {
   int i = 0; // Loop index

   // For each item call class member function to print
   for (i = 0; i < itemsInventory.size(); ++i) {
      cout << i << " - ";
      (*itemsInventory.at(i)).PrintItem();
   }
   return;
}

// Displays user commands supported by program
void PrintCommands() {
   cout << "Valid commands are: add, print, remove, quit" << endl;
   return;
}

int main() {
   vector<myItem*> itemsInventory; // Vector of myItem pointers
   string productName;             // Name of item in inventory
   int productQuantity = 0;        // Quantity of item in inventory
   string userInput;               // User command
   int listPos = 0;                // Position of itme in vector

   myItem* newItem = 0;            // Pointer used to create an item
   myItem* tmp = 0;                // Pointer used to lookup an item

   // Output user options
   PrintCommands();

   while (userInput != "quit") {
      // Prompt user for input
      cout << endl << "Your command: ";
      cin >> userInput;

      if (userInput == "add") {         // Add new item name/qty to vector
         cout << "   New item name: ";
         cin >> productName;
         cout << "   New item quantity: ";
         cin >> productQuantity;
         newItem = new myItem(productName, productQuantity);
         itemsInventory.push_back(newItem);
      }
      else if (userInput == "print") {  // Print current item name/qty in vector
         PrintAllItems(itemsInventory);
      }
      else if (userInput == "remove") { // Remove item from vector
         cout << "   List position number: ";
         cin >> listPos;
         if (listPos < itemsInventory.size()) {
```

```
            cout << "      Removed item " << listPos << "." << endl;
            tmp = itemsInventory.at(listPos);
            delete tmp;
            itemsInventory.erase(itemsInventory.begin() + listPos);
         }
         else {
            cout << "      Error removing: Item "
                 << listPos << " doesn't exist." << endl;
         }
      }
      else if (userInput != "quit"){    // Quit program
         PrintCommands();
      }
   }
   return 0;
}
```

```
Valid commands are: add, print, remove, quit

Your command: print

Your command: add
   New item name: shoes
   New item quantity: 16

Your command: add
   New item name: belt
   New item quantity: 33

Your command: print
0 - name: shoes, quantity: 16
1 - name: belt, quantity: 33

Your command: remove
   List position number: 0
      Removed item 0.

Your command: print
0 - name: belt, quantity: 33

Your command: quit
```

P Participation Activity

8.3.2: The new, delete, and -> operators.

#	Question	Your answer
1	Define a variable named "orange" as a pointer to class Fruit.	= 0;
2	Write a statement that allocates memory for the new variable "orange" that points to class Fruit. Do not use parentheses.	
3	For a variable named orange, write a statement that calls the member function RemoveSeeds that returns void and accepts no parameters. Use the -> operator.	
4	Write a statement to deallocate memory pointed to by variable orange, which is a pointer to class Fruit.	

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

Challenge Activity

8.3.1: Allocating memory

Allocate memory for houseHeight using the new operator.

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

int main() {
   double* houseHeight = 0;

   /* Your solution goes here  */

   *houseHeight = 23;
   cout << "houseHeight is " << *houseHeight;

   return 0;
}
```

Run

C Challenge Activity

8.3.2: Deallocating memory

Deallocate memory for kitchenPaint using the delete operator.

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

class PaintContainer {
   public:
      ~PaintContainer();
      double gallonPaint;
};

PaintContainer::~PaintContainer() { // Covered in section on Destructors.
   cout << "PaintContainer deallocated." << endl;
   return;
}

int main() {
   PaintContainer* kitchenPaint;

   kitchenPaint = new PaintContainer;
   kitchenPaint->gallonPaint = 26.3;
```

Run

Section 8.4 - String functions with pointers

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. Recall that the C string library must first be included via: `#include <cstring>`

String functions accept a char pointer for a string argument. That pointer is commonly a char array variable, or a string literal (each of which is essentially a pointer to the 0th element of a char array), but could also be an explicit char pointer. Example of such functions are strcmp(), strcpy(), and strchr(), introduced elsewhere.

Figure 8.4.1: String functions accept char pointers as arguments.

```
char string1[10] = "abcxyz";
char string2[10] = "xyz";
char newText[10] = "";
char* subStr = 0;

if (strcmp(string1, string2) == 0) {       // abcxyz does not equal xyz
   ...
if (strcmp(&string1[3], "xyz") == 0) {    // xyz equals xyz
   ...
subStr = &string1[3];                      // Points to 'x' in string1
if (strcmp(subStr, string2) == 0) {        // xyz equals xyz
   ...
strcpy(newText, subStr);                   // newText is now "xyz"
```

Table 8.4.1: Some C string modification functions.

Given:

```
char orgName[100] = "The Dept. of Redundancy Dept.";
char newText[100] = "";
char* subString = 0;
```

strchr()	`strchr(sourceStr, searchChar)` Returns 0 if searchChar does not exist in sourceStr. Else, returns pointer to first occurrence.	
strrchr()	`strrchr(sourceStr, searchChar)` Returns 0 if searchChar does not exist in sourceStr. Else, returns pointer to LAST occurrence (searches in reverse, hence middle 'r' in name).	
strstr()	`strstr(str1, str2)` Returns char* pointing to first occurrence of string str2 within string str1. Returns 0 if not found.	

strchr()

```
if (strchr(orgName, 'D') != 0) {     // 'D' exists in orgN
   subString = strchr(orgName, 'D'); // Points to first 'D
   strcpy(newText, subString);       // newText now "Dept.
}
if (strchr(orgName, 'Z') != 0) {     // 'Z' exists in orgN
   ... // Doesn't exist, branch not taken
}
```

strrchr()

```
if (strrchr(orgName, 'D') != 0) {    // 'D' exists in orgN
   subString = strrchr(orgName);     // Points to last 'D'
   strcpy(newText, subString);       // newText now "Dept.
}
```

strstr()

```
subString = strstr(orgName, "Dept"); // Points to first 'D
if (subString != 0) {
   strcpy(newText, subString);       // newText now "Dept.
}
```

The following example carries out a simple censoring program, replacing an exclamation point by a period and the word "Boo" by "---" (assuming those items are somehow bad and should be censored):

Figure 8.4.2: String searching example.

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

int main(void) {
   const int MAX_USER_INPUT = 100;      // Max input size
   char userInput[MAX_USER_INPUT] = ""; // User defined string
   char* stringPos = 0;                 // 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 != 0) {
      *stringPos = '.';
   }

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

   if (stringPos != 0) {
      strncpy(stringPos, "---", 3);
   }

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

   return 0;
}
```

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

Participation Activity

8.4.1: Modifying and searching strings.

#	Question	Your answer
1	Define a char* variable named charPtr.	
2	Assuming `char* firstR;` is already declared, store in firstR a pointer to the first instance of an 'r' in the char* variable userInput.	
3	Assuming `char* lastR;` is already declared, store in lastR a pointer to the last instance of an 'r' in the char* variable userInput.	
4	Assuming `char* firstQuit;` is already declared, store in firstQuit a pointer to the first instance of "quit" in the char* variable userInput.	

Challenge Activity

8.4.1: 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] = "Albert Johnson";
   char searchChar = 'J';
   char* searchResult = 0;

   /* Your solution goes here  */

   if (searchResult != 0) {
      cout << "Character found." << endl;
   }
   else {
      cout << "Character not found." << endl;
   }

   return 0;
```

Run

C Challenge Activity

8.4.2: 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] = "The Lion King";
   char* movieResult = 0;

   /* Your solution goes here  */

   cout << "Movie title contains The? ";
   if (movieResult != 0) {
      cout << "Yes." << endl;
   }
   else {
      cout << "No." << endl;
   }

   return 0;
```

Run

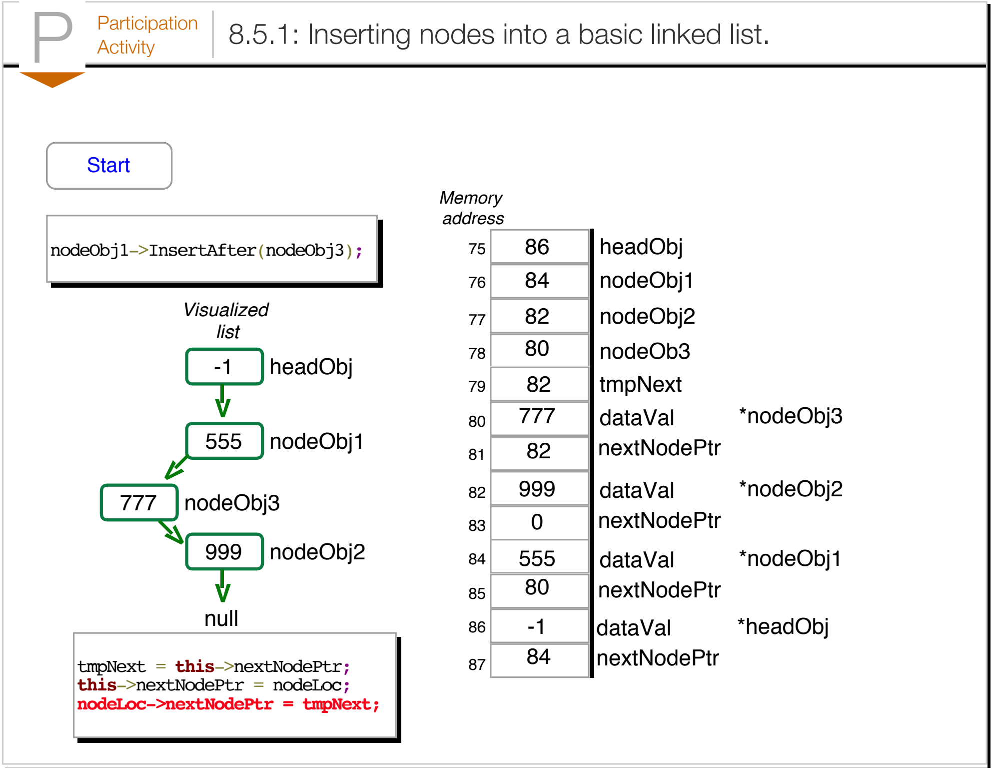
Section 8.5 - A first linked list

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

Figure 8.5.1: A basic example to introduce linked lists.

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

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

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

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
void IntNode::InsertAfter(IntNode* nodeLoc) {
   IntNode* tmpNext = 0;

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

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

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

int main() {
   IntNode* headObj  = 0; // Create intNode objects
   IntNode* nodeObj1 = 0;
   IntNode* nodeObj2 = 0;
   IntNode* nodeObj3 = 0;
   IntNode* currObj  = 0;

   // 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 != 0) {
      currObj->PrintNodeData();
      currObj = currObj->GetNext();
   }

   return 0;
}
```

```
-1
555
777
999
```



Participation Activity

8.5.1: Inserting nodes into a basic linked list.

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

Memory address

75	86	headObj	
76	84	nodeObj1	
77	82	nodeObj2	
78	80	nodeOb3	
79	82	tmpNext	
80	777	dataVal	*nodeObj3
81	82	nextNodePtr	
82	999	dataVal	*nodeObj2
83	0	nextNodePtr	
84	555	dataVal	*nodeObj1
85	80	nextNodePtr	
86	-1	dataVal	*headObj
87	84	nextNodePtr	

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

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

Participation Activity

8.5.2: A first linked list.

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

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

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

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

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

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

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

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
void IntNode::InsertAfter(IntNode* nodeLoc) {
   IntNode* tmpNext = 0;

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

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

// Grab location pointed by nextNodePtr
IntNode* IntNode::GetNext() {
   return this->nextNodePtr;
}
int main() {
   IntNode* headObj = 0; // Create intNode objects
   IntNode* currObj = 0;
   IntNode* lastObj = 0;
   int i = 0;            // 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 != 0) {
      currObj->PrintNodeData();
      currObj = currObj->GetNext();
   }

   return 0;
}
```

```
-1
1481765933
1085377743
1270216262
1191391529
812669700
553475508
445349752
1344887256
730417256
1812158119
147699711
880268351
1889772843
686078705
2105754108
182546393
1949118330
220137366
1979932169
1089957932
```

P Participation Activity

8.5.3: Managing a linked list.

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

```

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

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

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

Run

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

Exploring further:

- More on Linked Lists from cplusplus.com

Challenge Activity

8.5.1: 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 = 0);
   void InsertAfter(IntNode* nodePtr);
   IntNode* GetNext();
   int GetDataVal();
private:
   int dataVal;
   IntNode* nextNodePtr;
};

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

Run

Section 8.6 - Memory regions: Heap/Stack

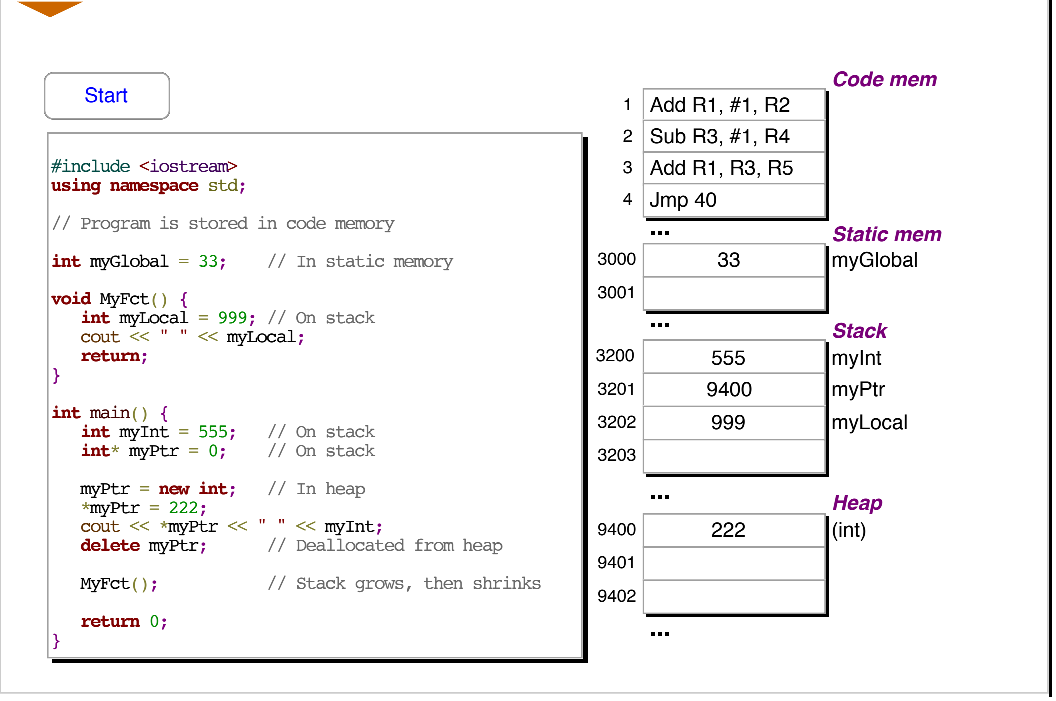
A program's memory usage typically includes four different regions:

- ***Code*** -- The region where the program instructions are stored.
- ***Static memory*** -- The region where global variables (variable defined outside any function) as well as static local variables (variables defined inside functions starting with the keyword "static") are allocated. The name "static" comes from these variables not changing (static means not changing); they are allocated once and last for the duration of a program's execution, their addresses staying the same.
- ***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***.

The following animation illustrates:

P Participation Activity

8.6.1: Use of the four regions of memory.

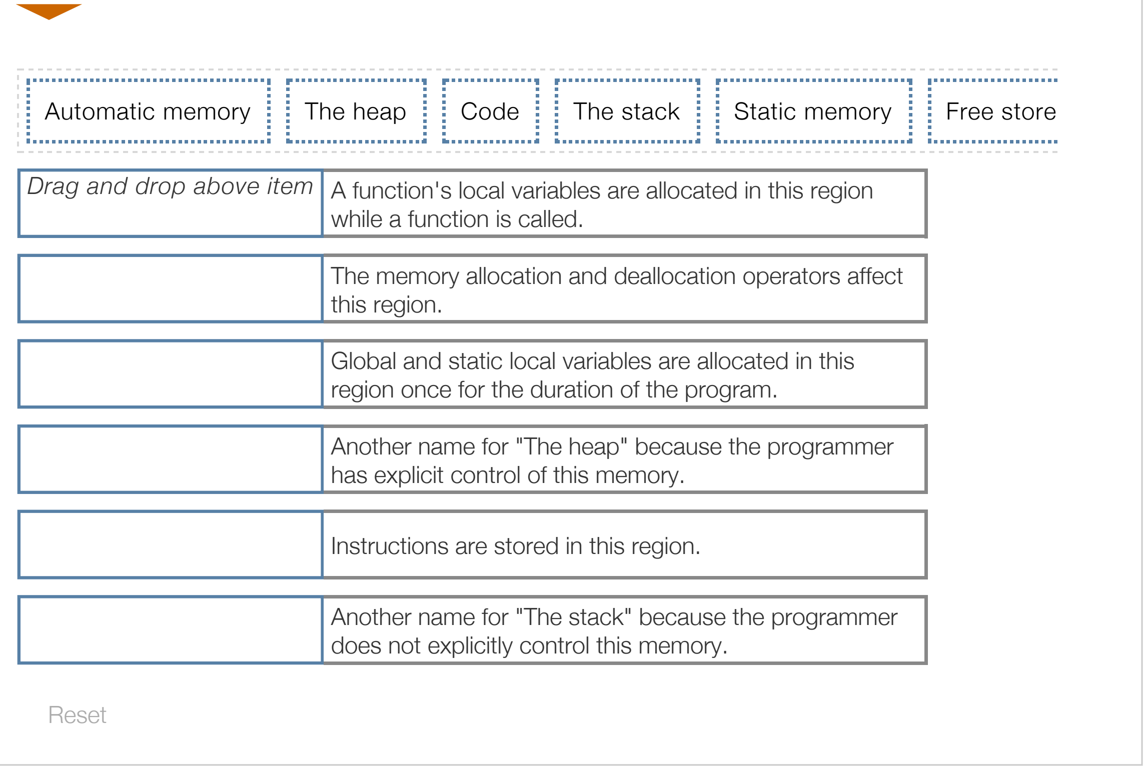


```
#include <iostream>
using namespace std;

// Program is stored in code memory

int myGlobal = 33;     // In static memory

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

int main() {
   int myInt = 555;    // On stack
   int* myPtr = 0;     // On stack

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

   MyFct();            // Stack grows, then shrinks

   return 0;
}
```

1	Add R1, #1, R2	
2	Sub R3, #1, R4	
3	Add R1, R3, R5	
4	Jmp 40	
...		
3000	33	myGlobal
3001		
...		
3200	555	myInt
3201	9400	myPtr
3202	999	myLocal
3203		
...		
9400	222	(int)
9401		
9402		
...		

P Participation Activity

8.6.2: Stack and heap definitions.



Drag and drop above item	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.

Section 8.7 - Miscellaneous pointer issues

Recall that each member function of a class has an implicit local variable named "this", and that data members in a member function can be accessed using the notation `this->dataMember`. The reason for the notation of "->" rather than "." to access the members should now be clear -- "this" is a pointer to the class object for which the member function is being called.

Pass by reference parameters closely resemble pointers. The compiler implements such parameters using pointers, but those details are hidden from the programmer. In the C language, pass by reference parameters don't exist, and thus the programmer must explicitly use pointers to achieve the goal of allowing a function to modify an argument:

- `void MyFct(int* inputParm) {   *inputParm = *inputParm + 1;   return; } // Fct definitio`
- `MyFct(&someValue); // Fct call`

The above code is valid for both C and C++. However, C++ introduced pass by reference parameters to avoid having to resort to pointers as above, and thus passing parameters as pointers can usually be avoided.

P Participation Activity

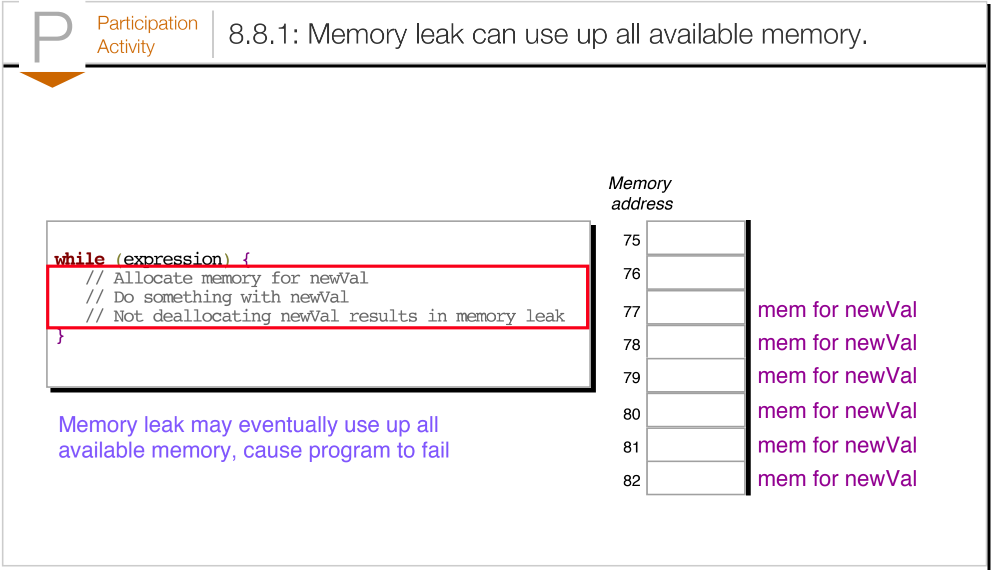
8.7.1: Pointer issues.

#	Question	Your answer
1	If myClass has data item num, then in a myClass member function, the statement x = num; is effectively the same as x = this.num;.	True False
2	The key advantage of pass by reference over passing pointers is faster access.	True False

---

Section 8.8 - Memory leaks

A program that allocates memory but then loses the ability to access that memory, typically due to failure to properly destroy/free dynamically allocated memory, is said to have a ***memory leak***. The program's available memory has portions leaking away and becoming 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. Such occupying of memory can slow program runtime, or worse can cause the program to fail if the memory becomes completely full and the program is unable to allocate additional memory. The following animation illustrates.



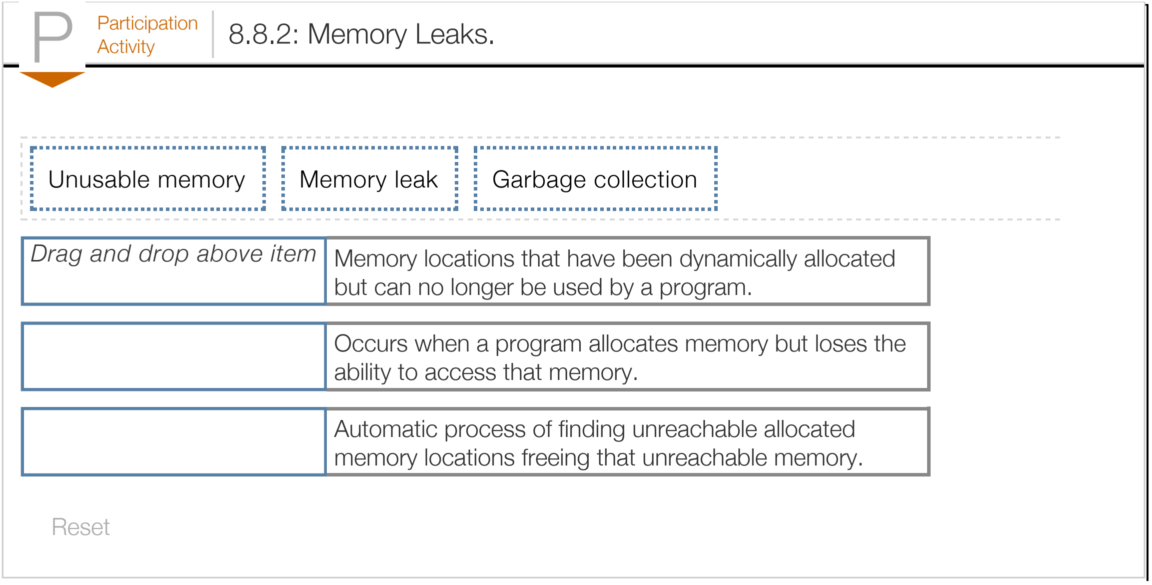
Participation Activity 8.8.1: Memory leak can use up all available memory.

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

Memory leak may eventually use up all available memory, cause program to fail

Failing to free allocated memory when done using that memory, resulting in a memory leak, is a common error. Many programs that are commonly left running for long periods, such as web browsers, suffer from known memory leak problems -- just do a web search for "<your-favorite-browser> memory leak" and you'll likely find numerous hits.

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 freeing such unreachable memory. Some C/C++ implementations include garbage collection but those implementations are not standard. Garbage collection can reduce the impact of memory leaks, at the expense of runtime overhead. Computer scientists debate whether new programmers should learn to explicitly free memory versus letting garbage collection do the work.



Participation Activity 8.8.2: Memory Leaks.

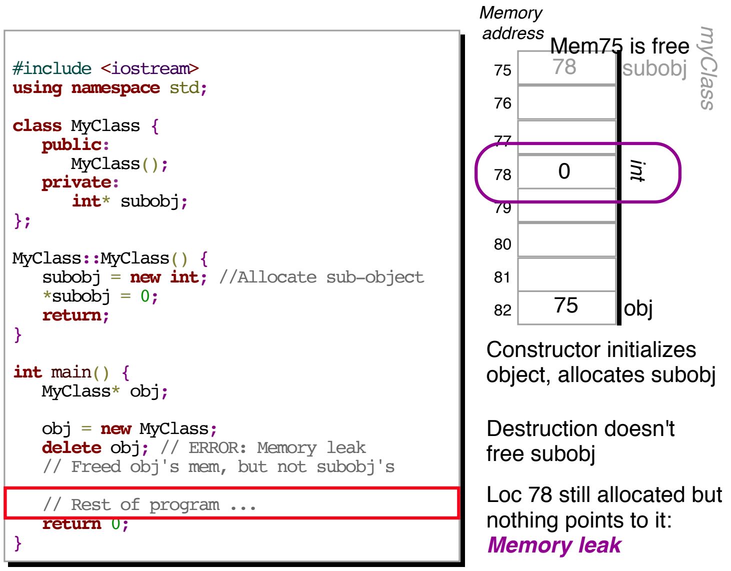
Drag and drop above item	Memory locations that have been dynamically allocated but can no longer be used by a program.
	Occurs when a program allocates memory but loses the ability to access that memory.
	Automatic process of finding unreachable allocated memory locations freeing that unreachable memory.

Section 8.9 - Destructors

Objects of a class type may be created and destroyed during a program's execution. For example, a program may create an object of type MyClass using the "new" operator, which allocates memory for the object, and later destroy that object using the "delete" operator, which frees that memory so it can be used again later by the program. The ***constructor*** function of a class automatically initializes the object's data members. Sometimes the converse is necessary, namely a function that is automatically called when an object is destroyed, known as a ***destructor***.

Destructors are needed when destroying an object should involve more work than simply freeing the object's memory. Such a need commonly arises when an object's data member, which we'll call a sub-object, had additional allocated memory. Freeing the object's memory without also freeing the sub-object's memory results in a problem wherein the sub-object's memory is no longer accessible but can't be used again by the program. The following animation illustrates.

Participation Activity 8.9.1: Lack of destructor function yields memory leak.



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

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

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

int main() {
   MyClass* obj;

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

   // Rest of program ...
   return 0;
}
```

Constructor initializes object, allocates subobj

Destruction doesn't free subobj

Loc 78 still allocated but nothing points to it: ***Memory leak***

(Note that the program is very simple to focus on the main point. In particular, 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 that object.)

The `new myClass` operation allocates memory for a new myClass object and automatically calls myClass's constructor. While most constructors we've seen so far just initialize data members, myClass's constructor also *allocates memory for a sub-object*. The subsequent `delete myClass` operation just frees the memory of the myClass object, but does not also free the memory of the sub-object. That sub-object's memory is still allocated but is not pointed to by any object and is thus inaccessible.

The programmer, not the compiler, wrote the constructor function and thus knows that memory was allocated for a sub-object, and thus it makes sense that the programmer should also write a destructor function that frees that sub-object's memory.

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 (not even void). The syntax for declaring and defining a destructor for a class named "myClass" is:

Construct 8.9.1: Class Destructor.

```
class myClass {
  public:
     ...
     ~myClass();
     ...
};
myClass::~myClass() {
  ...
}
```

The following is simple class example with a destructor function. The destructor frees the sub-object's memory that was allocated by the constructor.

Figure 8.9.1: Simple class with a destructor.

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

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

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

MyClass::~MyClass() {
   cout << "Destructor called." << endl;
   if (dataObj != 0) {
      delete dataObj;
   }
   return;
}

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

   tempClassObj = new MyClass; // Allocate mem for object
   delete tempClassObj;        // No more memory leak
                               // Freed obj's mem, including dataObj

   // Rest of program ...
   return 0;
}
```

```
Constructor called.
Destructor called.
```

A destructor is typically needed whenever a constructor acquires resources. One common example of acquiring resources is when the constructor allocates memory, as above, in which case the destructor might free that memory. Another example is when a constructor opens a file, in which case the destructor might close that file.

Destroying an object occurs when the object:

- *Goes out of scope* -- The object was allocated as a local object for a function or other block, and that function is now returning.
- *Is deleted* -- The object was allocated using the "new" operator and is now being deleted using the "delete" operator.

Destroying an object involves several steps that include:

1. *Calling the object's destructor* -- The object's destructor function is called first. Not defining a destructor is equivalent to defining a destructor with no statements.
2. *Calling the destructor functions for each member* of the object.
3. *Calling the destructor function for each base class* of the object.

After the above, the object's own memory is freed.

Earlier examples involving classes did not include destructor functions, yet did not contain memory leaks. The reason is because those classes' members either were basic types like int, whose memory is part of the class object itself and that memory is freed with the object, or were abstract data types like string or vector whose destructors get called automatically when the object is destroyed (per step 2 above).

Participation Activity

8.9.2: Destructors.

#	Question	Your answer

1	Declare a destructor for class `engineMap`.	`engineMap::` ____ `() {`
		`// destructor function code`
		`}`

2	Define a destructor for a class mp3Info that deletes songTitle then returns.	`mp3Info::~mp3Info() {` ____ `return;` `}`
3	List the order in which the IntNode and TwoIntsNode destructors will be called after the statement delete myNode;. In your answer, use the class names only separated by commas.	____

3

```
class IntNode {
public:
   IntNode(int data = 0,
IntNode* next = 0);
   ~IntNode();
   ...
};

class TwoIntsNode {
public:
   IntNode a;
   IntNode b;

   TwoIntsNode();
   ~TwoIntsNode();
   ...
};

int main() {
   TwoIntsNode myNode =
new TwoIntsNode();
   ...
   delete myNode;
   ...
}
```

Exploring further:

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

C Challenge Activity

8.9.1: Write a destructor

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

```
Destroying CarCounter
```

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

class CarCounter {
   public:
      CarCounter();
      ~CarCounter();
   private:
      int carCount;
};

CarCounter::CarCounter() {
   carCount = 0;
   return;
}

/* Your solution goes here  */

int main() {
```

Run

Section 8.10 - Copy constructors

In the following code, main() creates tempClassObj of type MyClass; note from the output that the MyClass constructor is automatically called. main() then sets and prints a value for the data member dataObj, the value being 9. So far so good.

main() then calls SomeFunction(), where tempClassObj is passed by value, which creates a local copy of the argument. When SomeFunction() returns, the local copy of the object goes out of scope and MyClass' destructor is automatically called. main() then prints tempClassObj's dataObj value again, which should have been 9, but is actually printed as 0.

Can you determine the problem?

Figure 8.10.1: Problem that can occur without copy constructor.

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

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

   // Set member value dataObj
   void SetDataObj(const int setVal) {
      *dataObj = setVal;
   }

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

// Default constructor
MyClass::MyClass() {
   cout << "Constructor called." << endl;
   dataObj = new int; // Allocate mem for data
   *dataObj = -1;
   return;
}

// Destructor
MyClass::~MyClass() {
   cout << "Destructor called." << endl;
   if (dataObj != 0) {
      delete dataObj;
   }
   return;
}

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

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

   // Set and print data member value
   tempClassObj.SetDataObj(9);
   cout << "Before: " << tempClassObj.GetDataObj() << endl;

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

   // Print data member value
   cout << "After: " << tempClassObj.GetDataObj() << endl;
   return 0;
}
```

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

The problem is that the local object copy automatically made during the call to SomeFunction() merely copied the pointer to tempClassObj's dataObj, rather than making a copy of the dataObj. When SomeFunction() returns, while the local object was being destroyed, its destructor freed the dataObj's memory, which was also being pointed to by tempClassObj. That memory no longer belonged to tempClassObj, and in this case happened to have gotten changed to 0 but may have stayed at 9 or been changed to another number.

Furthermore, note when main() returns, tempClassObj going out of scope causes the destructor to be called again, which tries to free tempClassObj's dataObj's memory but results in an error message (only part of which is shown) because that memory was already freed.

The solution is to create a new constructor that will be automatically called when a function call creates a local copy of an object, that constructor makes a new copy of the dataObj, known as a ***deep copy*** of the object. Recall that a ***constructor*** for a class is a special member function that is automatically called when an object of that class is created, the function initializing the object's members. Recall also ***overloading*** means to give multiple functions the same name but different parameter/return types, the compiler determining which function to call by a call's types. Among possible constructors for a class, two kinds are known by particular names:

- The ***default constructor*** can be called with no arguments. So that constructor might have no parameters, or may have parameters all with default values.

- 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. If the programmer doesn't define a copy constructor, then the compiler implicitly defines one with statements that perform a memberwise copy, which simply copies each member using assignment: `newObj.memberVal1 = origObj.memberVal1, newObj.memberVal2 = origObj.memberVal2`, etc. That behavior works fine for many classes, but typically a deep copy is instead desired for a member that is a pointer.

Therefore, a programmer may define their own copy constructor, typically having the form:

Construct 8.10.1: Copy constructor.

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

A class's copy constructor will be called automatically when an object of the class type is passed by value to a function, and also when an object is initialized by copying another object during definition, as in: `MyClass classObj2 = classObj1; or obj2Ptr = new MyClass(classObj1);`.

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

Figure 8.10.2: Problem solved by creating a copy constructor that does a deep copy.

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

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

   // Set member value dataObj
   void SetDataObj(const int setVal) {
      *dataObj = setVal;
   }

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

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

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

// Destructor
MyClass::~MyClass() {
   cout << "Destructor called." << endl;
   if (dataObj != 0) {
      delete dataObj;
   }
   return;
}

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

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

   // Set and print data member value
   tempClassObj.SetDataObj(9);
   cout << "Before: " << tempClassObj.GetDataObj() << endl;

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

   // Print data member value
   cout << "After: " << tempClassObj.GetDataObj() << endl;
   return 0;
}
```

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

Note that the above discussion uses a trivially-simple class having a dataObj whose type was just 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.

P Participation Activity

8.10.1: Determining which constructor will be called.

Given the following class declaration and variable definition, 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;
```

#	Question	Your answer

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

C Challenge Activity

8.10.1: Write a copy constructor.

Write a copy constructor for CarCounter that assigns origCarCounter.carCount to the constructed object'
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() {
   carCount = 0;
```

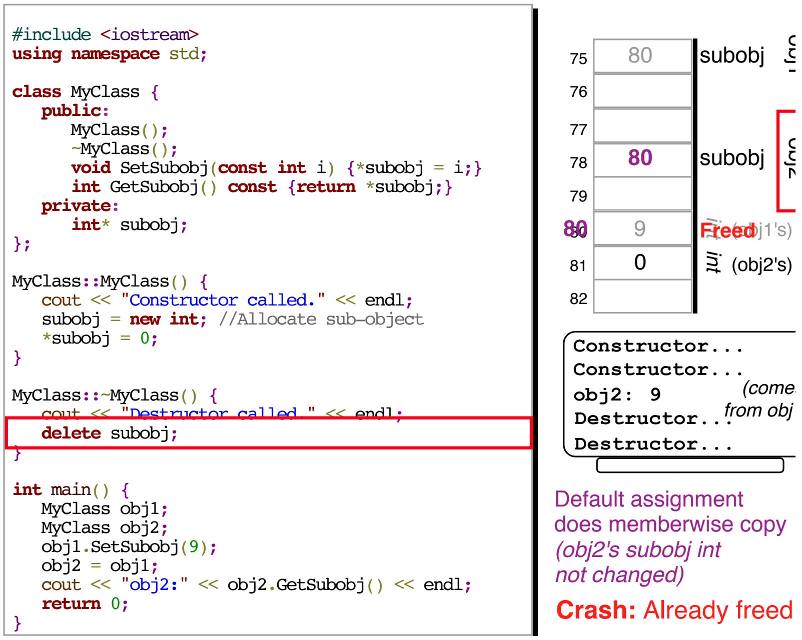
Run

Section 8.11 - Copy assignment operator

Sometimes a programmer wishes to copy one already-created object to another already-created object. For example, given two MyClass objects classObj1 and classObj2, a programmer might write `classObj2 = classObj1;`. The default behavior of the assignment operator "=" for classes or structs is to perform memberwise assignment, i.e., `classObj2.memberVal1 = classObj1.memberVal1`, `classObj2.memberVal2 = classObj1.memberVal2`, etc. Such behavior may work fine for members having basic types like int or char, but typically is not the desired behavior for a pointer member. The following animation illustrates a problem that can arise, for a trivially-simple class having just one member, which is an integer pointer (most classes have more members, and a pointer member would typically be to a more complex type than just an int).

P Participation Activity

8.11.1: Basic assignment operation fails when pointer member involved.



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

class MyClass {
   public:
      MyClass();
      ~MyClass();
      void SetSubobj(const int i) {*subobj = i;}
      int GetSubobj() const {return *subobj;}
   private:
      int* subobj;
};

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

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

int main() {
   MyClass obj1;
   MyClass obj2;
   obj1.SetSubobj(9);
   obj2 = obj1;
   cout << "obj2:" << obj2.GetSubobj() << endl;
   return 0;
}
```

```
Constructor...
Constructor...
obj2:  9
Destructor...
Destructor...
```

Default assignment does memberwise copy *(obj2's subobj int not changed)*

Crash: Already freed

The problem is that the assignment of `classObj2 = classObj1;` merely copied the pointer for dataObj, resulting in classObj1's dataObj and classObj2's dataObj members both pointing to the same memory location. Printing classObj2 prints 9 but for the wrong reason, and if classObj1's dataObj value was later changed, classObj2's dataObj value would seemingly magically change too. Additionally, destroying classObj1 frees that dataObj's memory; destroying classObj2 then tries to free that same memory, causing a program crash. Furthermore, a memory leak has occurred because neither dataObj is pointing at location 81.

The solution is to overload the "=" operator by defining a new function, known as the ***copy assignment operator*** or sometimes just the ***assignment operator***, that copies one class object to another. Such a function is typically defined as:

Construct 8.11.1: Overloading the class copy assignment operator.

```
class MyClass {
   public:
      ...
      MyClass& operator=(const MyClass& objToCopy);
      ...
};
```

The syntax may look odd but that's how a function can be defined to overload the assignment operator "=". The new assignment function should properly copy members, including allocating new memory for pointer members, known as a ***deep copy***. The following program solves the above problem by introducing an assignment operator.

Figure 8.11.1: Assignment operator performs a deep copy.

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

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

   // Set member value dataObj
   void SetDataObj(const int setVal) {
      *dataObj = setVal;
   }

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

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

// Destructor
MyClass::~MyClass() {
   cout << "Destructor called." << endl;
   if (dataObj != 0) {
      delete dataObj;
   }
   return;
}

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

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

   return *this;
}

int main() {
   MyClass tempClassObj1; // Create object of type MyClass
   MyClass tempClassObj2; // Create object of type MyClass

   // Set and print object 1 data member value
   tempClassObj1.SetDataObj(9);

   // Copy class object using copy assignment operator
   tempClassObj2 = tempClassObj1;

   // Set object 1 data member value
   tempClassObj1.SetDataObj(1);

   // Print data values for each object
   cout << "obj1:" << tempClassObj1.GetDataObj() << endl;
   cout << "obj2:" << tempClassObj2.GetDataObj() << endl;

   return 0;
}
```

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

Participation Activity

8.11.2: Assignment operator.

#	Question	Your answer
1	Declare a copy assignment operator for a class named `engineMap` using `inVal` as the input parameter name.	`engineMap& operator=( );`
2	Provide the return statement for the copy assignment operator for the engineMap class.	

Challenge Activity

8.11.1: Write a copy constructor.

Write a copy assignment operator for CarCounter that assigns objToCopy.carCount to the new objects's
Sample output for the given program:

```
Cars counted: 12
```

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

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

CarCounter::CarCounter() {
   carCount = 0;
```

Run

Section 8.12 - Rule of three

We have seen that classes have several special member functions. One of them is:

- **Default constructor**: A constructor is a class member function that is automatically called immediately after memory is allocated for an object. The constructor's job is to initialize the object's members. A ***default constructor*** is a version of a constructor that can be invoked without arguments. The default constructor is automatically called when an object is defined as in `MyClass obj;` or allocated via the new operator as in

`objptr = new MyClass;`.
    - If the programmer doesn't define a default constructor for a class, the compiler implicitly defines one having no statements, meaning the constructor does nothing.

Good practice is always initializing variables, a programmer should similarly be explicit in defining a default constructor for a class, making sure to initialize each member.

Additional special member functions are:

- **Destructor**: A ***destructor*** is a class member function that is automatically called when an object of the class is destroyed, as when the object goes out of scope or is explicitly destroyed as in `delete obj;`.
    - If the programmer doesn't define a destructor for a class, the compiler implicitly defines one having no statements, meaning the destructor does nothing.

- **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 localObj)` and the call `SomeFunction(anotherObj)`, when an object is initialized when defined such as `MyClass classObj1 = classObj2;`, or when an object is initialized when allocated via "new" as in `obj1Ptr = new MyClass(classObj2);`.
    - 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., `classObj2.memberVal1 = classObj1.memberVal1`, `classObj2.memberVal2 = classObj1.memberVal2`, etc.

- **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.
    - If the programmer doesn't define a copy assignment operator, the compiler implicitly defines one that does a memberwise copy.

For each of those three special member functions, the implicitly-defined behavior is often sufficient. However, for some cases such as when a class has a pointer member and the default constructor allocates memory for that member, then the programmer likely needs to explicitly define the behavior for all three of those special member functions.

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 (destructor, copy constructor, copy assignment operator) if any one of these functions are defined.

P Participation Activity

8.12.1: Rule of three.

#	Question	Your answer
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 prevObj` has already been defined, the statement `MyClass obj2 = prevObj;` will call the copy assignment operator.	True False

5	Assuming `MyClass prevObj` has already been defined, the following variable definition will call the copy assignment operator.
	`MyClass obj2;`
	`...`
	`obj2 = prevObj;`
		True
		False

Exploring further:

- More on Rule of Three from Wikipedia.

Section 8.13 - C++ example: Employee list using vectors

P Participation Activity

8.13.1: Managing an employee list using a vector.

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

1. Run the program, and provide input to add three employees' names and related data. Then use the list option to display the list.
2. Modify the program to implement the deleteEntry function.
3. Run the program again and add, list, delete, and list again various entries.

Reset

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

```

```
a
Rajeev Gupta
Sales
ager
```

Run

Below is a solution to the above problem.

P Participation Activity

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

Reset

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

```
a
Rajeev Gupta
Sales
nager
```

Run