13.1 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

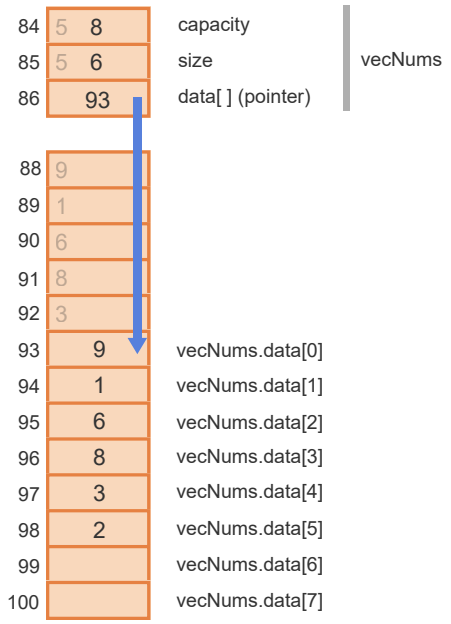
The C++ vector class is a container that internally uses a ***dynamically allocated array***, an array whose size can change during runtime. When a vector is created, the vector class internally dynamically allocates an array with an initial size, such as the size specified in the constructor. If the 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 13.1.1: Dynamically allocated arrays.

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

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

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



Animation content:

Static figure:
Begin Cpp code:

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

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

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

End Cpp code.
```


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

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

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

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

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

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

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

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

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

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

Animation captions:

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

PARTICIPATION ACTIVITY 13.1.2: 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.

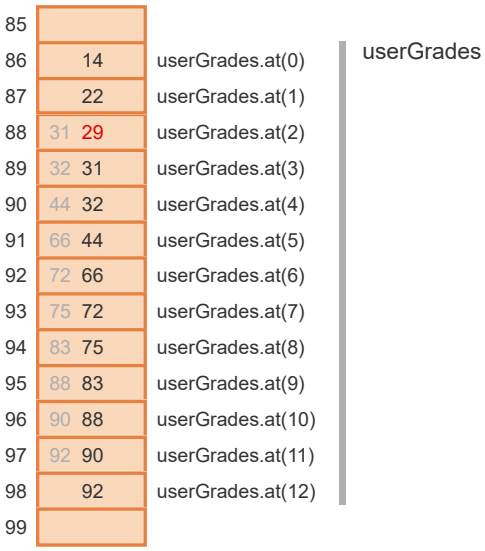
- True
- False

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 13.1.3: Vector insert performance problem.

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



Animation content:

Static figure:
Begin Cpp code:

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

End Cpp code.

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

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

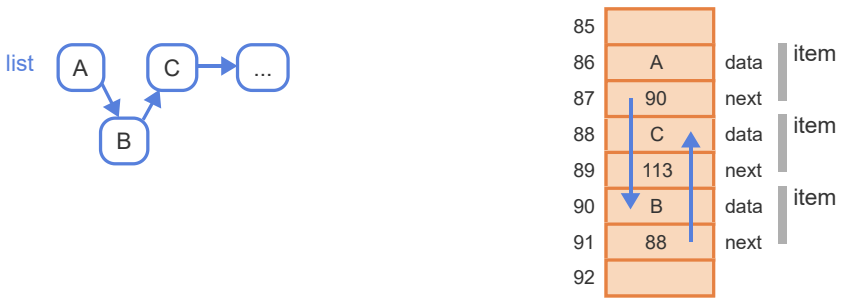
The line of code, userGrades.insert(userGrades.begin() + 2, 29);, is highlighted and the values from memory addresses 88 to 97 shift one memory address to memory addresses 89 to 98. The value, 29, is highlighted in the line of code, userGrades.insert(userGrades.begin() + 2, 29);, and moves to memory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

Animation captions:

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

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

PARTICIPATION ACTIVITY 13.1.4: A list avoids the shifting problem.



Animation content:

Static figure:

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

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

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

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

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

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

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

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

Animation captions:

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

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

Table 13.1.1: 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 13.1.5: 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

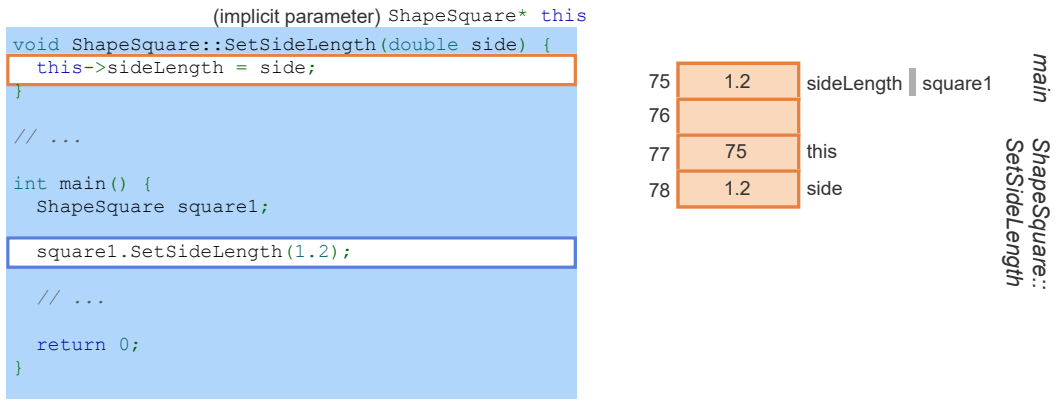
7) Finding the 500th item in a 999-item linked list requires visiting how many items? Correct answer is one of 0, 1, 500, and 999.

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 13.1.6: Pointers used to call class member functions.



Animation content:

Static figure:
Begin Cpp code:

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

```
// ...

int main() {
  ShapeSquare square1;

  square1.SetSideLength(1.2);

  // ...

  return 0;
}
```


End Cpp code.

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

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

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

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

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

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

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

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

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

Animation captions:

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

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

PARTICIPATION ACTIVITY 13.1.7: 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

13.2 Pointer basics

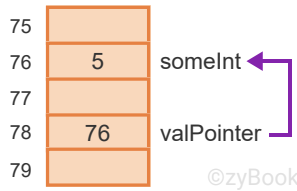
Pointer variables

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 13.2.1: Assigning a pointer with an address.

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

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

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

   return 0;
}
```




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

Animation content:

Static figure:
Begin Cpp code:

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

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

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

  return 0;
}
```

End Cpp code.

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

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

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

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

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

unknown value.

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

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

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

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

Animation captions:

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

Printing memory addresses

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

PARTICIPATION ACTIVITY | 13.2.2: 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.

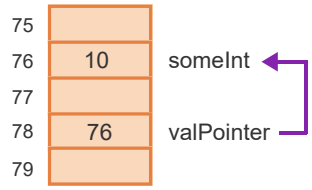
Check Show answer

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 13.2.3: Using the dereference operator.

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

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

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

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

   *valPointer = 10;   // Changes someInt to 10

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

   return 0;
}
```




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

Animation content:

Static figure:
Begin Cpp code:

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

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

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

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

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

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

   return 0;
}
```

End Cpp code.

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

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

The lines of code,

```
int someInt;
int* valPointer;

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

valPointer = &someInt;
```

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

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

someInt address is 76
valPointer is 76,

are output to the console.

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

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

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

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

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

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

Animation captions:

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

PARTICIPATION ACTIVITY 13.2.4: 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

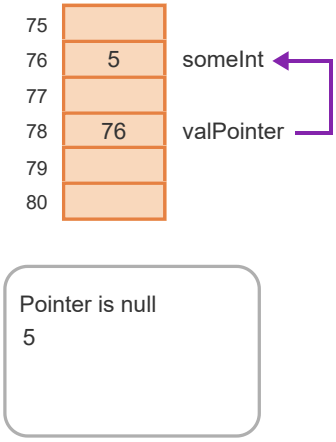
Null pointer

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 13.2.5: Checking to see if a pointer is null.


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

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

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

   return 0;
}
```



Animation content:

Static figure:
Begin Cpp code:

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

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

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

  return 0;
}
```

End Cpp code.

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

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

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

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

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

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

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

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

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

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

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

Animation captions:

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

Null pointer

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

PARTICIPATION ACTIVITY 13.2.6: Null pointer.

Refer to the animation above.

1) The code below outputs 3.

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

- True
- False

2) The code below outputs 5.

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

- True
- False

3) The code below outputs 7.

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

- True
- False

Common pointer errors

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

Common syntax errors:

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

Common runtime errors:

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

PARTICIPATION ACTIVITY | 13.2.7: Common pointer errors.

syntax errors

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

*valPointer = &someInt;

valPointer = &someInt;
```

✖ int value cannot be assigned int*

◀ remove *

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

int* valPointer1;
int* valPointer2;
```

✖ int cannot be assigned nullptr

◀ declare on separate lines

runtime errors

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

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

✖ dereferencing unknown address

◀ initialize pointer before dereferencing


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

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

✖ dereferencing a null pointer

◀ initialize null pointer to valid address before dereferencing

Animation content:

Static figure:
Begin syntax error Cpp code 1:

int someInt = 5;
int* valPointer;

*valPointer = &someInt;

valPointer = &someInt;

End Cpp code.

Begin syntax error Cpp code 2:

valPointer1 = nullptr;
valPointer2 = nullptr;

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

End Cpp code.

Begin runtime error Cpp code 1:

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

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


End Cpp code.

Begin runtime error Cpp code 2:

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

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

End Cpp code.

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

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

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

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

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

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


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

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

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

red. The lines of code,

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

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

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

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

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

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

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

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

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

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

Animation captions:

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

PARTICIPATION ACTIVITY 13.2.8: 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 13.2.1: Using pointers.

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

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

Load default template... Run

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     double vehicleMpg;
6     double* valPointer = nu
7
8     valPointer = &vehicleMp
9
10    *valPointer = 29.6; //
11                        //
12
13    // vehicleMpg = 40.0;
14
15    cout << "Vehicle MPG =
16    cout << "Vehicle MPG =
17
18
```

CHALLENGE ACTIVITY 13.2.1: Enter the output of pointer content.

519134.1787752.qx3zqy7

Start

Type the program's output

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

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

   someNumber = 8;
   numberPointer = &someNumber;

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

   return 0;
}
```

```
8 8
```


1 2 3

Check Next

CHALLENGE ACTIVITY 13.2.2: Printing with pointers.

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

```
Items: 990
```

Learn how our autograder works

519134.1787752.qx3zqy7

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int* numItemsPointer;
6     int numItems;
7
8     cin >> numItems;
9
10    /* Your solution goes here  */
11
12    if (numItemsPointer == nullptr) {
13       cout << "Items is negative" << endl;
14    }
15    else {
16       cout << "Items: " << *numItemsPointer << endl;
17    }
18
```

Run

View your last submission

CHALLENGE ACTIVITY 13.2.3: Pointer basics.

519134.1787752.qx3zqy7

Start

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

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

Ex: If the input is `18.5 13 L`, then the output is:

```
Tide level: 18.5 meters
Recorded at hour: 13
Alert: L
```



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

int main() {
   double height;
   int time;
   char alert;

   /* Your code goes here */

   cin >> height;
   cin >> time;
   cin >> alert;

   cout << "Tide level: " << fixed << setprecision(1) << *heightPointer << " meters" << endl;
   cout << "Recorded at hour: " << *timePointer << endl;
   cout << "Alert: " << *alertPointer << endl;
```

1 2 3

Check Next level

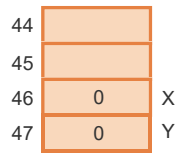
13.3 Operators: new, delete, and ->

The new operator

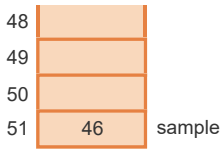
The ***new operator*** allocates memory for the given type and returns a pointer to the allocated memory. If the type is a class, the new operator calls the class's constructor after allocating memory for the class's member variables.

PARTICIPATION ACTIVITY | 13.3.1: The new operator allocates space for an object, then calls the constructor.


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

class Point {
public:
   Point();

   double X;
   double Y;
};
```



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

   X = 0;
   Y = 0;
}

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



Console:

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

Animation content:

Static figure:
Begin Cpp code:

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

class Point {
public:
  Point();

  double X;
  double Y;
};

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

  X = 0;
  Y = 0;
}

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

End Cpp code.

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

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

The lines of code,

double X;
double Y;,

are highlighted.

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

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

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

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

X = 0;
Y = 0;,

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

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

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

Animation captions:

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

PARTICIPATION ACTIVITY 13.3.2: 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 13.3.3: Constructor arguments.

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

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

   double X;
   double Y;
};

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

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

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

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

   return 0;
}
```

Console:

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

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

Animation content:

Static figure:
Begin Cpp code:

#include <iostream>
using namespace std;

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

  double X;
  double Y;
};

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

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

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

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

   return 0;
}
```

End Cpp code.

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

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

The lines of code,

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

are highlighted.

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

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

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

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

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

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

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

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

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

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

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

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

Animation captions:

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

PARTICIPATION ACTIVITY 13.3.4: Constructor arguments.

If unable to drag and drop, refresh the page.


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

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

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

```
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 13.3.1: 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 13.3.5: 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`

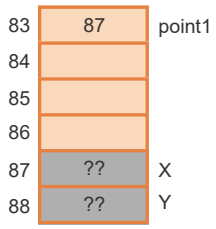
The delete operator

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 13.3.6: The delete operator.

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

   delete point1;

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



Console:

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

Animation content:

Static figure:
Begin Cpp code:

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

  delete point1;

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

End Cpp code.

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

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

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

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

are highlighted and the lines,

X = 73
Y = 19

are output to the console.

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

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

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

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

Animation captions:

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

PARTICIPATION ACTIVITY 13.3.7: The delete operator.

1) The delete operator can affect any pointer.
   - ○ True
   - ○ False

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

- True
- False

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

- True
- False

Allocating and deleting object arrays

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

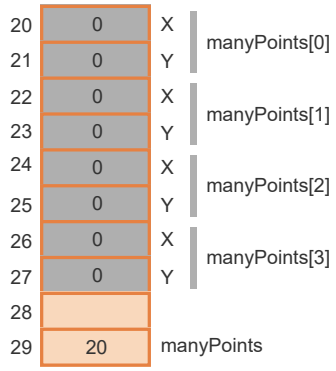
The ***delete[] operator*** is used to free an array allocated with the new operator.

PARTICIPATION ACTIVITY 13.3.8: Allocating and deleting an array of Point objects.

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

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

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

   return 0;
}
```

Console:

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



Animation content:

Static figure:
Begin Cpp code:

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

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

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

  return 0;
}
```

End Cpp code.

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

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

The lines of code,

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

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

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

The lines of code,

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


are highlighted. The line,

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

are output to the console.

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

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

Animation captions:

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

PARTICIPATION ACTIVITY | 13.3.9: 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 | 13.3.1: Using the new, delete, and -> operators.

519134.1787752.qx3zqy7

Start

Type the program's output

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

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

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

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

   myCar1 = new Car(70);
   myCar2 = new Car(80);

   return 0;
}
```

```
Traveled: 70
Traveled: 80
```


1 2 3

Check Next

CHALLENGE ACTIVITY 13.3.2: Deallocating memory

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

Learn how our autograder works

519134.1787752.qx3zqy7

```
1  #include <iostream>
2  using namespace std;
3
4  class PaintContainer {
5     public:
6        ~PaintContainer();
7        double gallonPaint;
8  };
9
10 PaintContainer::~PaintContainer() { // Covered in section on Destructors.
11    cout << "PaintContainer deallocated." << endl;
12 }
13
14 int main() {
15    PaintContainer* kitchenPaint;
16
17    kitchenPaint = new PaintContainer;
18    kitchenPaint->gallonPaint = 26.3;
```

Run

View your last submission

CHALLENGE ACTIVITY 13.3.3: Operators: new, delete, and ->.

519134.1787752.qx3zqy7

Start

Two integers are read as the age and the weight of a Goat object. Assign pointer myGoat with a new Goat object using the a and the weight as arguments in that order.

Ex: If the input is `7 59`, then the output is:

```
Goat's age: 7
Goat's weight: 59
```

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

class Goat {
   public:
      Goat(int ageValue, int weightValue);
      void Print();
   private:
      int age;
      int weight;
};
Goat::Goat(int ageValue, int weightValue) {
   age = ageValue;
   weight = weightValue;
}
void Goat::Print() {
   cout << "Goat's age: " << age << endl;
   cout << "Goat's weight: " << weight << endl;
```


1 2 3 4 5

Check Next level

Exploring further:

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