8.1 Why pointers?

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

Vectors use dynamically allocated arrays

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

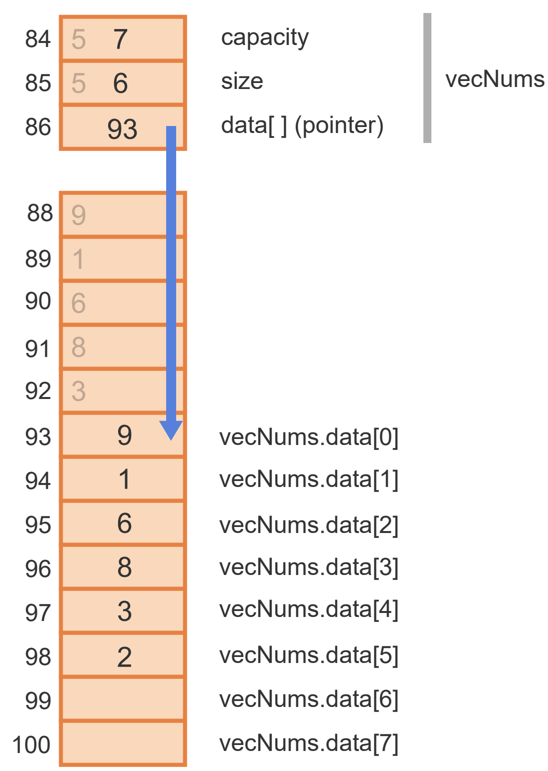
The ability to dynamically change the size of a vector makes vectors more powerful than arrays. Built-in constructs have also made vectors safer to use in terms of memory management.

PARTICIPATION ACTIVITY 8.1.1: Dynamically allocated arrays.

Start 2x speed

```
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
```



Feedback?

PARTICIPATION ACTIVITY | 8.1.2: Dynamically allocated arrays.

1) The size of a vector is the same as the vector's capacity.

- True
- False

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

- True
- False

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

- True
- False

Feedback?

Inserting/erasing in vectors vs. linked lists

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

PARTICIPATION ACTIVITY | 8.1.3: Vector insert performance problem.

Start 2x speed

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

85		
86	14	userGrades.at(0)
87	22	userGrades.at(1)
88	31 29	userGrades.at(2)
89	32 31	userGrades.at(3)
90	44 32	userGrades.at(4)
91	66 44	userGrades.at(5)
92	72 66	userGrades.at(6)
93	75 72	userGrades.at(7)
94	83 75	userGrades.at(8)
95	88 83	userGrades.at(9)
96	90 88	userGrades.at(10)
97	92 90	userGrades.at(11)
98	92	userGrades.at(12)
99		

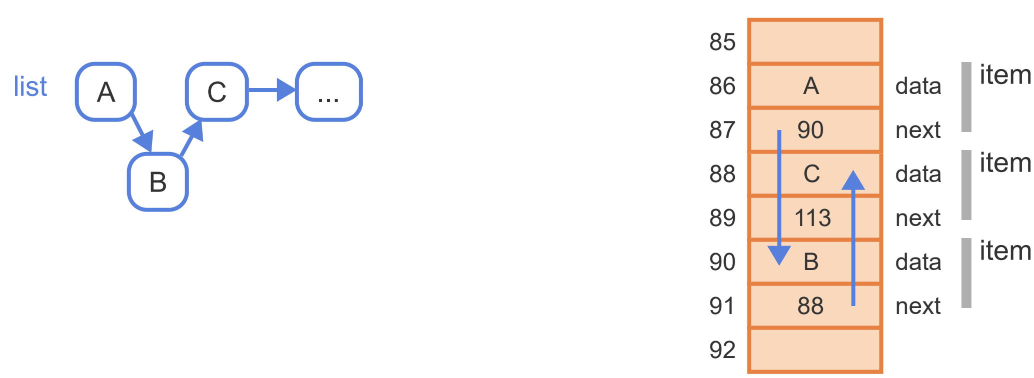
userGrades

Feedback?

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

PARTICIPATION ACTIVITY 8.1.4: A list avoids the shifting problem.

Start 2x speed



Feedback?

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

Table 8.1.1: Comparing vectors and linked lists.

<table>
<tr><th>Vector</th><th>Linked list</th></tr>
<tr><td>

Stores items in contiguous memory locations
Supports quick access to i'th element via <code>at(i)</code>
May be slow for inserts or erases on large lists due to necessary shifting of elements

</td><td>

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

</td></tr>
</table>

Feedback?

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

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

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

0

Check Show answer

Correct

0

Appending just places the new item at the end of the vector. No shifting of existing elements is necessary.

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

Check Show answer

Answer

999

Inserting at the front requires making room for the new element. So every current element must be shifted once.

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

0

Check Show answer

Correct

0

Deleting from the end just removes the item from the end. No gap is created in the middle of the vector, so no shifting is needed.

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

999

Check Show answer

Answer

998

After the first item is deleted, 998 elements remain. Each must be shifted, in order to fill the gap of the deleted item.

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

0

Check Show answer

Correct

0

The new item is simply added to the end.

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

0

Check Show answer

Correct

0

No shifting of other items is required, which is an advantage of using pointers.

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.

500

Check Show answer

Correct

500

The program starts at the 1st item, follows that item's pointer to the 2nd item, follows that item's pointer to the 3rd item, ..., until reaching the 500th item. The inability to immediately access any data item is a drawback of linked lists.

Feedback?

Pointers used to call class member functions

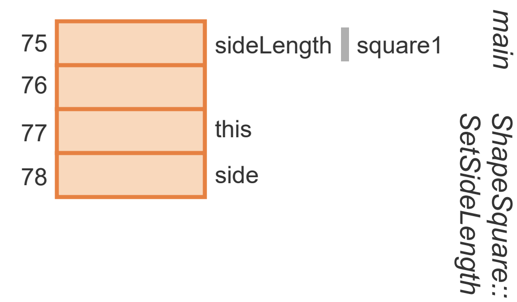
When a class member function is called on an object, a pointer to the object is automatically passed to the member function as an implicit parameter called the ***this*** pointer. The `this` pointer enables access to an object's data members within the object's class member functions. A data member can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

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

1 2 3 4 2x speed

(implicit parameter) `ShapeSquare* this`

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

// ...

int main() {
  ShapeSquare square1;

  square1.SetSideLength(1.2);

  // ...

  return 0;
}
```



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.

Feedback?

PARTICIPATION ACTIVITY 8.1.7: The 'this' pointer.

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

1) In SetFilmLength(), which would assign the data member filmLength with the value 120?

- (selected) this->filmLength = 120;
- this.filmLength = 120;
- 120 = this->filmLength;

Correct

'->' is the member access operator for pointers, and 'this' is a pointer referencing the current object.

2) In SetFilmLength(), which would assign the data member filmLength with the parameter filmLength?

- filmLength = filmLength;
- this.filmLength = filmLength;
- (selected) this->filmLength = filmLength;

Correct

The this->filmLength refers to the FilmInfo object's data member.

Feedback?

Exploring further:

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