Data Structures and Algorithms

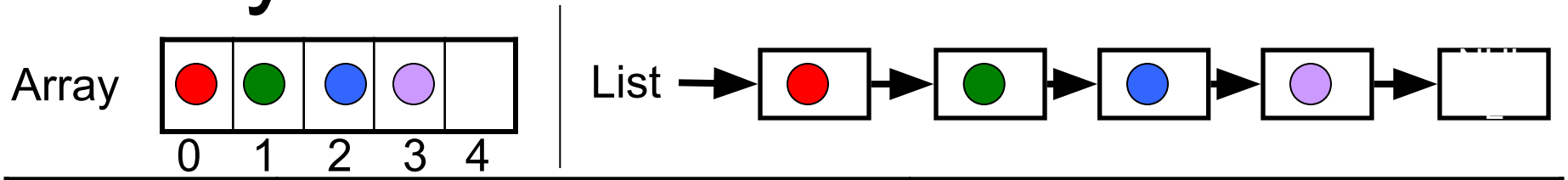
MichiganEngineering

Contents

- Elementary Containers
- Amortized Complexity
- Strings
- Valgrind
- Programming Questions

Elementary Containers

Arrays vs. Linked Lists



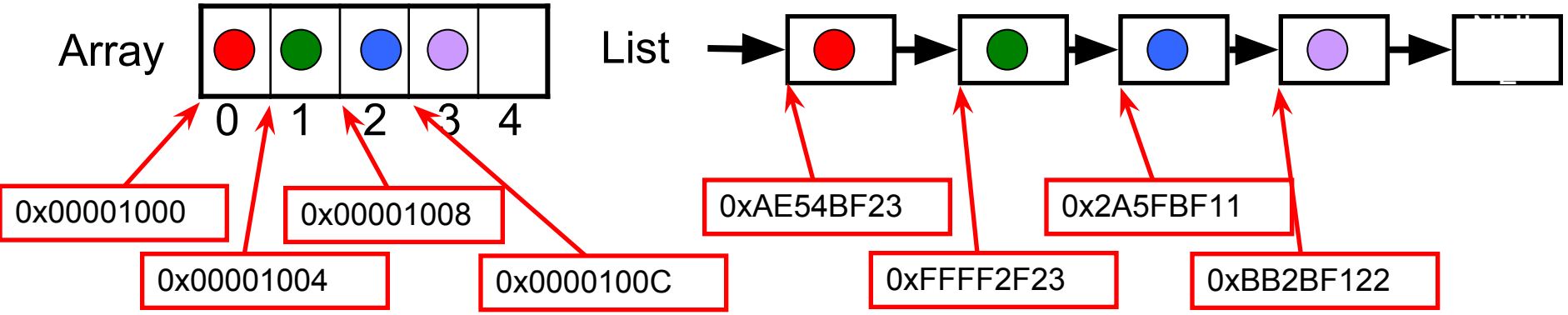
	Arrays	Linked Lists
Access	Random in O(1) time Sequential in O(1) time	Random in O(n) time Sequential in O(1) time
Insert Append	Inserts in O(n) time Appends in O(n) time (O(1) amortized possible if vector)	Inserts in O(n) time Appends in O(n) time
Bookkeeping	Ptr to beginning CurrentSize or ptr to end of space used (optional) MaxSize or ptr to end of allocated space (optional)	Size (optional) Head ptr to first node Tail ptr to last node (optional) In each node, ptr to next node Wasteful for small data items
Memory	Wastes memory if size is too large Requires reallocation if too small	Allocates memory as needed Requires memory for pointers

Worst Case

Worst Case

Arrays vs. Linked Lists

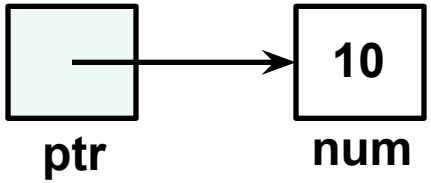
- Memory Layout
  - Arrays are allocated in contiguous chunks in memory
  - Linked lists are allocated in non-contiguous chunks in memory
- Impact
  - Pointer arithmetic only works with arrays



Arrays and Pointers

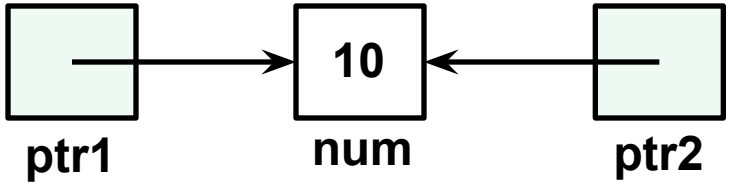
- Pointers store address locations

```
int num = 10;
int* ptr = &num;
```



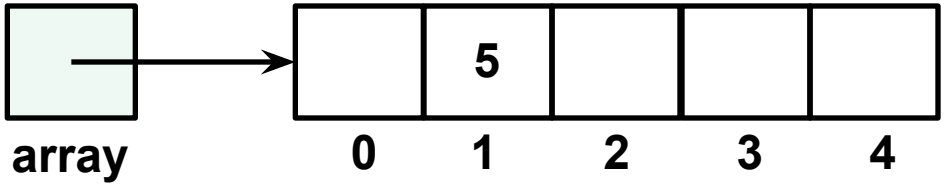
- Multiple pointers to one address location

```
int num = 10;
int* ptr1 = &num;
int* ptr2 = ptr1;
```



- Arrays == Pointers

```
int* array = new int[5];
array[1] = 5;
cout << array[1] << " " << *(array+1); // what does this output?
```



Arrays vs. Lists: cases

Which one is better at the following:

- Given a pointer to an element, insert a new element after
- Given an index, update an arbitrary element
- Search on sorted container
- Bulk-storing a known number of elements

Arrays vs. Lists: Final Comment

For projects:

- Prefer using vectors over C arrays
- Avoid pointers, and use STL containers instead
- Prefer vector<T> over list<T>
  - STL's optimizations favor vectors
    - Test it yourself!

Amortized Complexity

Amortized Complexity

- Amortized analysis looks at the complexity of an operation over multiple iterations.
- Not the same as average complexity!
- Average complexity gives an estimate of complexity that can be affected by "randomness"
- Amortized complexity is guaranteed over multiple iterations.

Array Resizing

- Arrays achieve O(1) random access since they occupy memory **contiguously**.
  - To access arr[5], add 5*sizeof(object) to arr
- When arrays run out of space, they need to be resized in order to continue inserting elements. Assuming that reallocating n objects of memory is an O(n) operation, **what would be the overall complexity of inserting 8 objects?**
  - Reallocation: `new` array with new size, copy contents of old array into new array, `delete[]` old array. In this case we are allocating a new array with new_size = size_of_old_array + 1

Array Resizing

- Arrays achieve O(1) random access since they occupy memory **contiguously**.
  - To access arr[5], add 5*sizeof(object) to arr
- When arrays run out of space, they need to be resized in order to continue inserting elements. Assuming that reallocating n objects of memory is an O(n) operation, **what would be the overall complexity of inserting 8 objects?**
  - Reallocation: `new` array with new size, copy contents of old array into new array, `delete[]` old array

Answer: 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36

MichiganEngineering

Array Resizing

- Reallocating space every time an element is inserted is O(n^2) over the course of inserting n elements. Additionally, we have O(n) inserts. We can do better!
- Dynamically resizing arrays (vector<T>) "double" in size every time they run out of space.
- With this, what's the cost of inserting 8 elements?

Array Resizing

- Reallocating space every time an element is inserted is O(n^2) over the course of inserting n elements. Additionally, we have O(n) inserts. We can do better!
- Dynamically resizing arrays (vector<T>) "double" in size every time they run out of space.
- With this, what's the cost of inserting 8 elements?

Answer: 1 + 2 + 4 + 8 = 15

Array Resizing: Final Comment

- Strings, Vectors, and other "auto-resizing" containers may reallocate their memory and move their data
  - When this happens, pointers to their data (such as those from .c_str()) are invalidated!
  - ie, if we store a pointer to vec[10] and the vector resizes, then the pointer is invalidated.

Strings

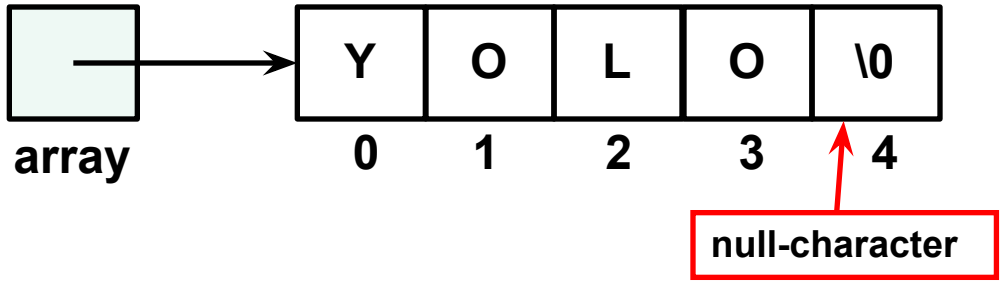
C-String

- char str[] = "HELLO\n";
  - What is the length of this C-string?

C-String

- Array of characters terminated by null-character

```
const char* array = "YOLO";
```
Or
```
char array[] = "YOLO";
```
Or
```
char* array = new char[5];
array[0] = 'y';
array[1] = 'o';
array[2] = 'l';
array[3] = 'o';
array[4] = '\0';
```



- C's Standard Library has many functions for C-Strings (<cstring> in C++)
  - Copying: `strcpy, strncpy`
  - Concatenation: `strcat, strncat`
  - Comparison: `strcmp, strncmp, etc.`
  - Searching: `strchr, strcspn, etc.`
  - Other: `strlen, strerror`

C++ String

- Full-fledged object, defined in <string> of STL

```
string str1 = "YOLO";       // str1 contains "YOLO"
string str2("YOLO");        // str2 contains "YOLO"
string str3("YOLO", 2, 1); // str3 contains "L"
```

- STL has many member functions for C++ strings
  - Iterators: `begin, end, etc.`
  - Capacity: `size, length, resize, reserve, clear, etc.`
  - Element Access: `operator[], at, back, front`
  - Modifiers: `operator+=, append, push_back, etc.`
  - Operations: `c_str, get_allocator, find, etc.`

MichiganEngineering

C-String vs. C++ String

- Use of functions on strings
  - C-String (function call with string passed as function argument)

```
char str[] = "YOLO";
cout << strlen(str); // prints 4
```

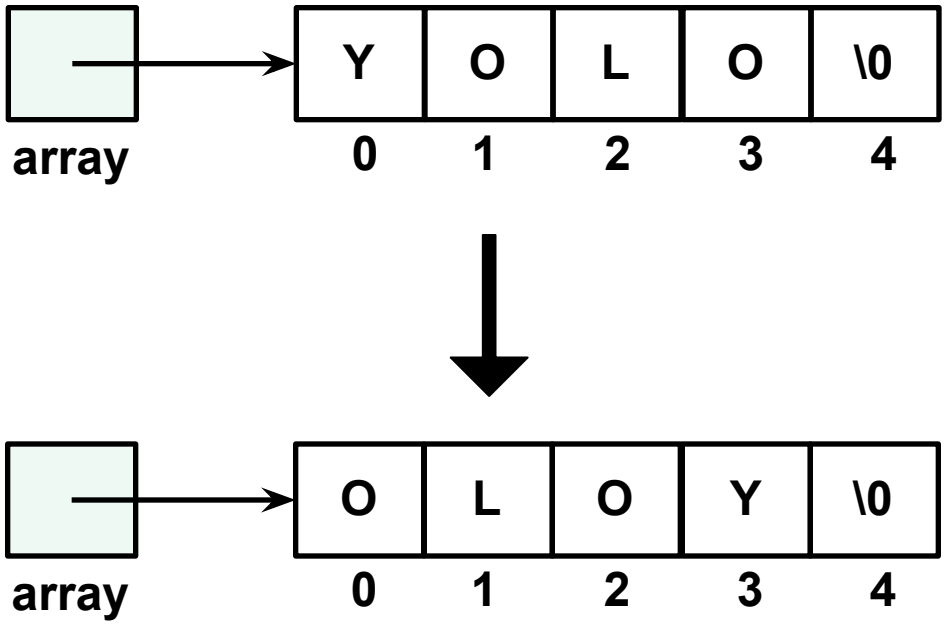
  - C++ String (dot operator for member function)

```
string str("YOLO");
cout << str.length(); // prints 4
```

- **In general, use C++ String over C-String**
  - Encapsulation
    - Size is stored - no need to keep track of null-terminator.
  - More fully featured: iterators, easy growth syntax, safety with getline, to_string, works cleanly with items in the algorithm header
  - Some C++ string functions are faster than C-String counterpart
    - Example: strlen() vs. .length()

POP QUIZ

- Reverse a C-String in place (you cannot use a separate buffer).



STL

Programming Questions

Problem 1

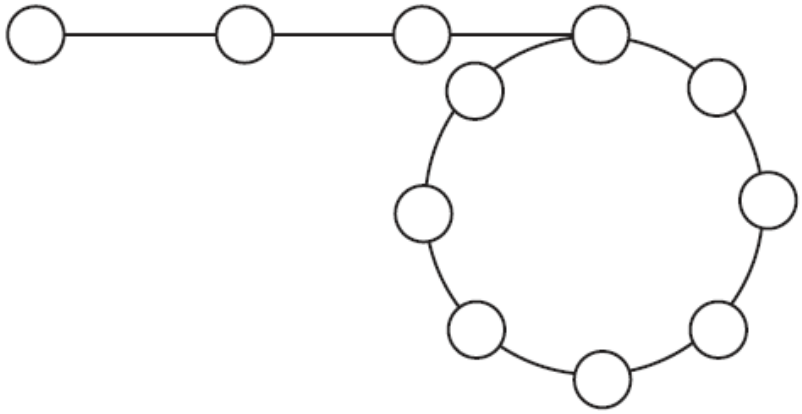
- Write a program to check if a string is a palindrome.
  - ex. "racecar"
  - bool isPalindrome(char* str); //C-string
  - bool isPalindrome(string str); //string

Problem 2

- Implement an algorithm to find the nth to last element of a singly linked list
  - node* nth_to_last(node* root, int n);
  - struct node{ int elt; node* next; };

Problem 3 (a little trickier!)

- Given a circular linked list, give an algorithm that returns a pointer to the node at the beginning of the loop
  - node* findLoopStart(node* root);



Valgrind Review

Valgrind: Motivation

- You run your code and see "`Segmentation Fault`"
- Memory bugs are difficult to find!
- Examples:
  - Using uninitialized memory
  - Writing/reading off an end of an array
  - Forgetting to free/delete pointers

Valgrind: Introduction

- A memory debugging/profiling tool
  - A focus on memcheck, a memory checking tool
- Typical bugs to find with Valgrind
  - Memory leaks
  - Double free/delete
  - Accessing non-allocated memory
  - Using uninitialized variables

Valgrind: Example Code

```
int main()
{
    vector<int> foo = {1, 2, 3};
    for (int i; i <= 3; i++)   // line 10
        cout << foo[i] << " "; // line 11
    cout << endl;
} // output: 1 2 3 0 (no seg fault)
```

MichiganEngineering

Valgrind: Example Output

```
==26655== Conditional jump or move depends on uninitialised value(s)
==26655==at 0x400BDD: main (test.cpp:10)
==26655==
==26655== Use of uninitialised value of size 8
==26655==at 0x400BBA: main (test.cpp:11)
==26655==
==26655== Invalid read of size 4
==26655==at 0x400BBA: main (test.cpp:11)
==26655==  Address 0x515304c is 0 bytes after a block of size 12 alloc'd
==26655==at 0x4A075FC: operator new(unsigned long) (vg_replace_malloc.c:298)
==26655==by 0x40116F: __gnu_cxx::new_allocator<int>::allocate(unsigned long, void const*)
      (new_allocator.h:104)
==26655==by 0x40105E: std::_Vector_base<int, std::allocator<int> >::_M_allocate(unsigned long) (in
      /tmp/student/a.out)
==26655==by 0x400EEF: void std::vector<int, std::allocator<int> >::_M_range_initialize<int const*>(int
      const*, int const*, std::forward_iterator_tag) (stl_vector.h:1201)
==26655==by 0x400D84: std::vector<int, std::allocator<int> >::vector(std::initializer_list<int>, std::
      allocator<int> const&) (stl_vector.h:368)
==26655==by 0x400B96: main (test.cpp:9)
```