CSCI 200: Foundational Programming Concepts & Design Lecture 35

Safe Programming: Exception Handling

Previously in CSCI 200

- Arrays
  - Stored in a one `n`-element contiguous block
  - Element access $O(1)$
  - All other operations $O(n)$
- Linked List
  - Stored in `n` one-element fragmented blocks
  - Element access $O(n)$
  - Some operations $O(1)$
  - Other operations $O(n)$

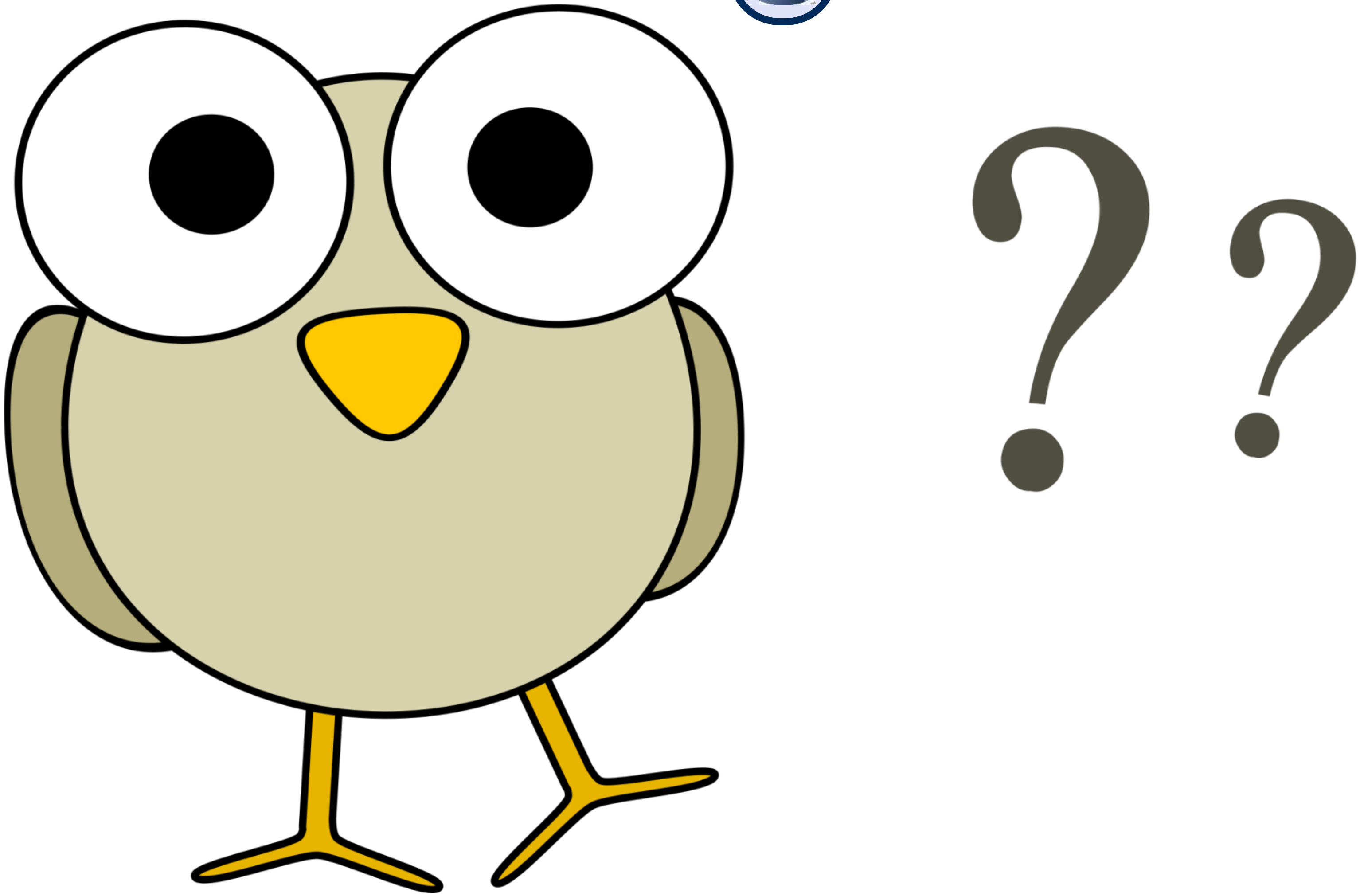
Data Structure Operations

Operation		Array	Singly-Linked List	Doubly-Linked List
Element Access		$O(1)$	$O(n)$	$O(n)$
Traversal	Forwards	$O(n)$	$O(n)$	$O(n)$
	Backwards		$O(n^2)$	$O(n)$
Add	Front	$O(n)$	$O(1)$	$O(1)$
	Middle		$O(n)$	$O(n)$
	Back		$O(1)$	$O(1)$
Delete	Front	$O(n)$	$O(1)$	$O(1)$
	Middle		$O(n)$	$O(n)$
	Back		$O(n)$	$O(1)$
Search		$O(n)$	$O(n)$	$O(n)$
Min / Max		$O(n)$	$O(n)$	$O(n)$
Memory		n*sizeof(T) contiguous	n*(sizeof(T)+8) fragmented	n*(sizeof(T)+16) fragmented

Previously in CSCI 200

- Linked List operations & Big O complexity
  - Be careful of dangling pointers!
  - Be careful of memory leaks!
  - Be careful of losing the reference to a node or start / end of the list!

Questions?



Learning Outcomes For Today

- Define what an exception is.
- Discuss why exceptions are thrown, how they are caught, and the benefits of using exceptions.
- Create a program that handles exceptions cleanly and prevents run time errors from occurring.

On Tap For Today

- Exception Handling
- Practice

On Tap For Today

- Exception Handling
- Practice

What Happens In Each Case?

```
int var1 = 999, var2 = 999;
int array[10];

cout << &var1 << " " << var1 << endl;
cout << &var2 << " " << var2 << endl;
cout << array << endl;

for(int i = -3; i <= 9; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl;
}

cout << var1 << endl;
cout << var2 << endl;
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
int var1 = 999, var2 = 999;
int array[10];

cout << &var1 << " " << var1 << endl; // prints 0x518 999
cout << &var2 << " " << var2 << endl; // prints 0x514 999
cout << array << endl;                // prints 0x520

for(int i = -3; i <= 9; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl;
}

cout << var1 << endl;
cout << var2 << endl;
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
int var1 = 999, var2 = 999;
int array[10];

cout << &var1 << " " << var1 << endl; // prints 0x518 999
cout << &var2 << " " << var2 << endl; // prints 0x514 999
cout << array << endl;                // prints 0x520

for(int i = -3; i <= 9; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl;
}

cout << var1 << endl;                 // prints -2
cout << var2 << endl;                 // prints -3
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
double var1 = 999, var2 = 999;
int array[10];

cout << &var1 << " " << var1 << endl;
cout << &var2 << " " << var2 << endl;
cout << array << endl;

for(int i = -6; i <= 9; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl;
}

cout << var1 << endl;
cout << var2 << endl;
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
double var1 = 999, var2 = 999;
int array[10];

cout << &var1 << " " << var1 << endl; // prints 0x510 999
cout << &var2 << " " << var2 << endl; // prints 0x508 999
cout << array << endl;                // prints 0x520

for(int i = -6; i <= 9; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl;
}

cout << var1 << endl;
cout << var2 << endl;
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
double var1 = 999, var2 = 999;
int array[10];

cout << &var1 << " " << var1 << endl; // prints 0x510 999
cout << &var2 << " " << var2 << endl; // prints 0x508 999
cout << array << endl;                // prints 0x520

for(int i = -6; i <= 9; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl;
}

cout << var1 << endl;                 // prints NaN --> exp = all 1s
cout << var2 << endl;                 // prints NaN
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
double var1 = 999, var2 = 999;
int array[10];

cout << &var1 << " " << var1 << endl;
cout << &var2 << " " << var2 << endl;
cout << array << endl;

for(int i = -100; i <= 100; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl;

}

cout << var1 << endl;
cout << var2 << endl;
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
double var1 = 999, var2 = 999;
int array[10];

cout << &var1 << " " << var1 << endl;
cout << &var2 << " " << var2 << endl;
cout << array << endl;

for(int i = -100; i <= 100; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl; // i = -14, bus error
                                                  // stack is corrupted
}

cout << var1 << endl;
cout << var2 << endl;
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
double var1 = 999, var2 = 999;
int *pArray = new int[10];

cout << &var1 << " " << var1 << endl;
cout << &var2 << " " << var2 << endl;
cout << array << endl;

for(int i = -10000000; i <= 10000000; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl;
}

cout << var1 << endl;
cout << var2 << endl;
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

What Happens In Each Case?

```
double var1 = 999, var2 = 999;
int *pArray = new int[10];

cout << &var1 << " " << var1 << endl;
cout << &var2 << " " << var2 << endl;
cout << array << endl;

for(int i = -10000000; i <= 10000000; i++) {
    array[i] = i;
    cout << &array[i] << " " << array[i] << endl; // seg fault
}

cout << var1 << endl;
cout << var2 << endl;
```

```
*results from
OS: macOS v12.1, Apple M1 chip
compiler: clang v12.0.5
target: arm64-apple-darwin21.2.0
your results may vary
```

Types of Access

- Read / Get

```
int arr[10], *pArr = new int[10], i;
cin >> i;
cout << arr[i] << " " << pArr[i] << endl;
```

- Write / Set

```
int arr[10], *pArr = new int[10], i, x;
cin >> i >> x;
arr[i] = x;
pArr[i] = x;
```

Abstract the Operation

- Read / Get

```
T get(const T * const P_ARRAY, const int SIZE, const int POS) {
  return P_ARRAY[POS];
}
```

- Write / Set

```
void set(T * const P_array, const int SIZE, const int POS, const T VAL) {
  P_array[POS] = VAL;
}
```

- No access protection still!

Abstract the Operation

- Read / Get

```
T get(const T * const P_ARRAY, const int SIZE, const int POS) {
  return P_ARRAY[POS];
}
```

- Write / Set

```
void set(T * const P_array, const int SIZE, const int POS, const T VAL) {
  if(POS >= 0 && POS < SIZE) {
    P_array[POS] = VAL;
  }
}
```

Abstract the Operation

- Read / Get

```
T get(const T * const P_ARRAY, const int SIZE, const int POS) {
 if(POS >= 0 && POS < SIZE) {
  return P_ARRAY[POS];
 } else {
  return ??? // what to do?
 }
}
```

- Write / Set

```
void set(T * const P_array, const int SIZE, const int POS, const T VAL) {
 if(POS >= 0 && POS < SIZE) {
  P_array[POS] = VAL;
 }
}
```

What Does vector/string Do?

```
vector<int> emptyVec;

emptyVec[5] = 5;
cout << emptyVec[-4] << endl;

emptyVec.at(5) = 5;
cout << emptyVec.at(-4) << endl;


string emptyStr;

emptyStr[5] = '?';
cout << emptyStr[-4] << endl;

emptyStr.at(5) = '?';
cout << emptyStr.at(-4) << endl;
```

What Does vector/string Do?

```
vector<int> emptyVec;

emptyVec[5] = 5;                   // seg fault
cout << emptyVec[-4] << endl;      // seg fault

emptyVec.at(5) = 5;                // exception std::out_of_range vector
cout << emptyVec.at(-4) << endl;   // exception std::out_of_range vector


string emptyStr;

emptyStr[5] = '?';                 // seg fault
cout << emptyStr[-4] << endl;      // seg fault

emptyStr.at(5) = '?';              // exception std::out_of_range basic_string
cout << emptyStr.at(-4) << endl;   // exception std::out_of_range basic_string
```

What's the difference?

- Seg Fault
  - Invalid memory access as reported by the OS resulting in a run time error
- Exception
  - **Thrown** programmatically in code by the program
  - Therefore, can **catch** the exception programmatically in code
  - If uncaught, results in a run time error

Throwing an Exception

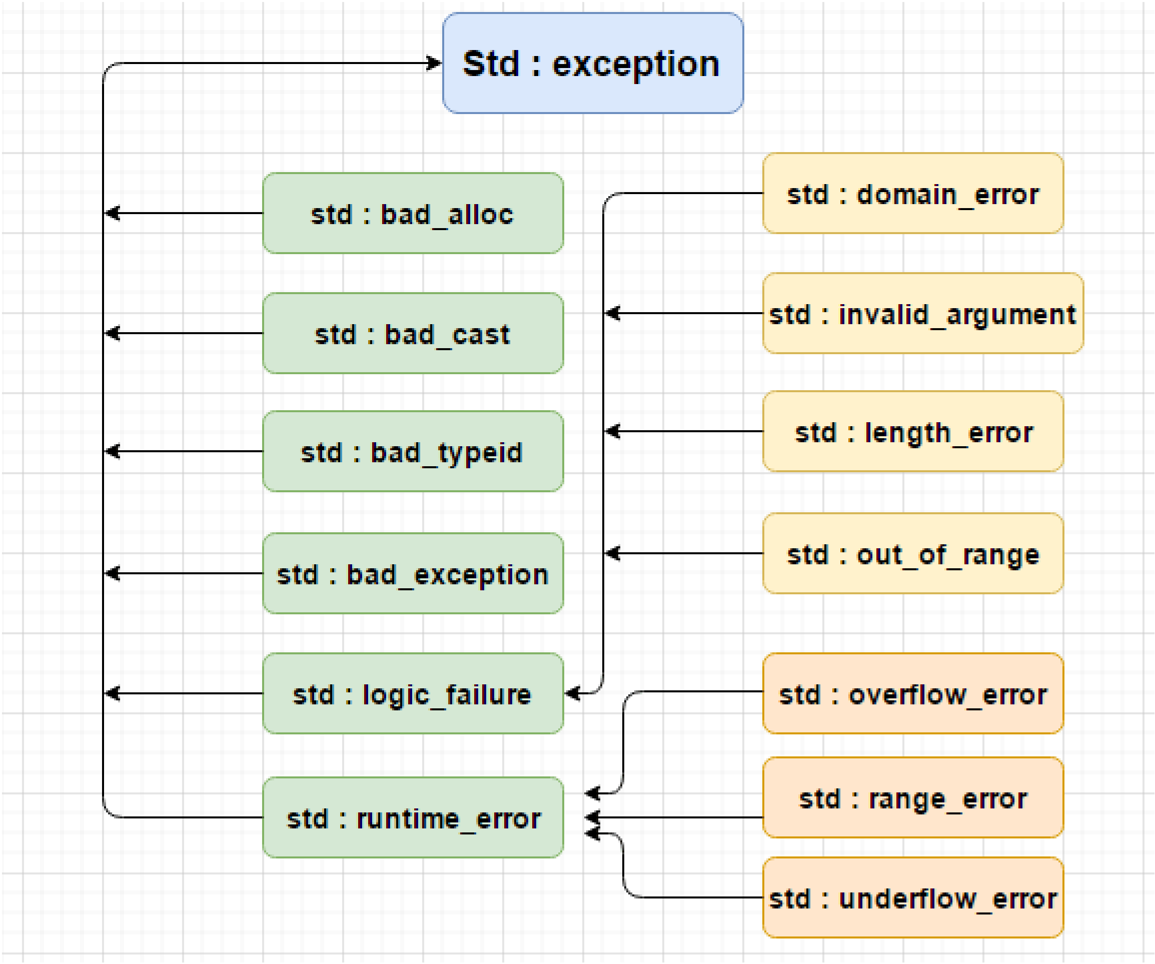
- Use the `throw` keyword to generate an exception
- Exceptions transfer control up the call stack
  - Halts execution of current stack frame
  - while call stack is not empty
    - If current stack frame does not handle exception, pops current stack frame and passes exception to next stack frame
    - If current stack frame handles exception, continues execution of current stack frame
  - If stack becomes empty, then run time error

Read / Get Operation

```
T get(const T * const P_ARRAY, const int SIZE, const int POS) {
  if(POS >= 0 && POS < SIZE) {
    return P_ARRAY[POS];
  } else {
    throw ??? // what to throw?
  }
}
```

- Need to throw a value which signals what type of exception has occurred
  - Have choices of what to do
    - Use standard exception type
    - Generate your own custom value

Standard Exceptions



Read / Get Operation

- Use standard type

```
T get(const T * const P_ARRAY, const int SIZE, const int POS) {
  if(POS >= 0 && POS < SIZE) {
    return P_ARRAY[POS];
  } else {
    string msg = "invalid access at pos " + to_string(POS)
                 + " for size " + to_string(SIZE);
    throw out_of_range(msg);
  }
}
```

- Use own value

```
const int INVALID_ACCESS = -5;
const int DIVIDE_BY_ZERO = -10;
T get(const T * const P_ARRAY, const int SIZE, const int POS) {
  if(POS >= 0 && POS < SIZE) {
    return P_ARRAY[POS];
  } else {
    throw INVALID_ACCESS;
  }
}
```

Catching Exceptions

- aka Exception Handling
  - Wrap code that may fail in a **try** block followed by a **catch** block for each type of exception that may occur

```
try {
  // statements that would throw an exception
} catch (ExceptionType1 e) {
} catch (ExceptionType2 e) {
} catch (...) { // generic catch anything that doesn't match above
}
```

Catching Exceptions

```
vector<int> myVec(5); // has 5 elements
for(int i = -1; i <= 5; i++) {
  try {
    cout << "accessing " << i << "...";
    myVec.at(i);
    cout << "succeeded!" << endl;
  } catch (out_of_range oore) {
    cerr << "out of range exception: " << oore.what() << endl;
  } catch (...) {
    cerr << "another exception occurred" << endl;
  }
}

/* output:

accessing -1...out of range exception: vector
accessing 0...succeeded!
accessing 1...succeeded!
accessing 2...succeeded!
accessing 3...succeeded!
accessing 4...succeeded!
accessing 5...out of range exception: vector

*/
```

Read / Get Operation

```
try {
  get(pArr, 5, -2)
} catch (out_of_range oore) {
  cerr << "out of range exception: " << oore.what() << endl;
} catch (int exceptionValue) {
  if(exceptionValue == INVALID_ACCESS) {
    cerr << "invalid array access" << endl;
  } else if(exceptionValue == DIVIDE_BY_ZERO) {
    cerr << "divide by zero error" << endl;
  }
} catch (...) {
  cerr << "something else happened that shouldn’t have" << endl;
}
```

Exception Handling

- **`try`** - **`throw`** - **`catch`** is a conscious choice by the developer to safely handle errors generated at runtime

On Tap For Today

- Exception Handling
- Practice

To Do For Next Time

- Can properly complete L6A

Inheritance + SOLID Quiz

- Make Canvas Full Screen
- Access Code:
- 12 Minutes