LECTURE 35: EXCEPTIONS AND ARRAYS

Today

- Reading
  - Weiss Ch. 8.1-8.3, Ch. 10
- Objectives
  - Exception handling in C++
  - Arrays in C++

C++ Exceptions

- Use try-catch block like Java
- In C++, you can throw a variable of any type

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

int main() {
    try{
        throw -1;
    }
    catch(int e) {
        cout << "Exception occurred: " << e << endl;
    }
    return 0;
}
```

type thrown and type caught must match

C++ Exceptions

- Can have multiple catch statements
- Use "…" to catch an exception of any type

```
try{
    // code here
}
catch(int e) {
    cout << "Integer exception: " << e << endl;
}
catch(char e) {
    cout << "Char exception: " << e << endl;
}
catch(...){
    cout << "Default exception" << endl;
}
```

Throw Lists

- Indicate what exceptions are thrown by function using throw lists

```
double sqrt(int x) throw(int);
```

- No list means throws anything (for backward compatibility)
- Empty list means throws nothing
- Throw lists are deprecated as of C++11 (but still supported)

C++ Exceptions

- C++ standard library includes `exception` class from which exception objects can be created
- The `exception` class includes the `what` function
  - Returns a `string` description
- All exceptions thrown by C++ standard library are derived from exceptions class
- See C++ documentation
  - `http://www.cplusplus.com/reference/exception/exception/`

Exceptions with File I/O

- Read this article:

http://gehrcke.de/2011/06/reading-files-in-c-using-ifstream-dealing-correctly-with-badbit-failbit-eofbit-and-perror/

Arrays in C++

- Prefer `vector` over using a primitive array
- Prefer `string` over using an array of characters
- Still, it's useful to understand primitive arrays in C++

Declaring an array

- Declare the type and size of the array

  int arr[3]; // notice where the brackets go

- Compiler allocates enough memory

  4 bytes per integer * 3 integers = 12 bytes allocated

- Use `sizeof()` function to get the size of a type in bytes

Behind the scenes

- The name of an array is a *constant* pointer to the beginning of the allocated memory for that array
- The pointer `arr` is guaranteed to equal `&arr[0]`

```
int main() {
   int SIZE = 3;
   int arr[SIZE];
   return 0;
}
```

Pointer Arithmetic

- You can perform addition on a pointer
- What is the value of ptr+1?
  - If ptr is an integer pointer, then adds 4 bytes to ptr
  - If ptr is a char pointer, then adds 1 byte to ptr
- Explains why arrays start with 0 instead of 1 in C-based languages

Implications

- The following is illegal in C++. Why?

```
int array1[3];
int array2[3];
array2 = array1;
```

Implications

- Saying array2 == array1 tests memory equality
- What happens when we pass an array as an input argument?

```
int main() {
    int arr[5];
    my_function(arr);
    return 0;
}
```

```
void function(int array[] ) {
    ...

}
```

Other differences

- There is no length instance variable

  arr.length; // doesn't work in C++

- Must keep track of the length of the array yourself
- No bounds checking in C++
  - Accessing beyond bounds of the array may result in segmentation fault…or may not

Dynamically allocated arrays

- Use the `new[ ]` operator
  - Just like the new operator but for arrays
  - creates an array of objects on the heap
  - There is a corresponding `delete[ ]` operator

```
void my_function() {
    int SIZE = 3;
    int array1[SIZE];              // allocated on the stack
    int *array2 = new int[SIZE];   // allocated on the heap
}
```

After my_function returns, what memory is freed and what is not?

Primitive strings

- An array of characters terminated with null terminator '\0'
- Any characters after null terminator ignored by string functions
- Can pass as type (char *)