Review

- C++ exception handling mechanism
  - Try-throw-catch block
    - How does it work
  - What is exception specification?
  - What if a exception is not caught?

Some useful C++ concepts and introduction to data structures

C++ programs with command line arguments

- int main(int argc, char* argv[]) {}
  - argc is the count of command line arguments. argc >=1. Command line arguments are separated by spaces.
  - argv is an array of pointers to character strings that contain the actual command-line arguments.
  - See sample1.c for the use of command line arguments.

C++ header files

- Source files use the *#include* directive to include the header files.
- Sometimes using header files can cause problems, see sample2.cpp.
  - Including a header file multiple times may cause "duplicate declaration" errors.
  - Why including stdio.h two times does not have any problem?
    - Look at /usr/include/stdio.h, this file is protected.

C++ header files

- The following mechanism prevents the body of a header file from being included multiple times.

```
#ifndef MYHEADER
#define MYHEADER
....
/* the body of the header file */
#endif
```

C++ macros with parameters

- Macros with parameters look/work like functions:
  - #define max(a, b) (a>b)?a:b

- Macros with parameters need to be defined carefully, otherwise weird things can happen.
  - What is wrong with the following macro?
    - #define sum(a, b) a+b

C++ macros with parameters

- What is wrong with the following macro?
  - #define sum(a, b) a +b
  - Checkout sample3.c

- How to fix the problem?

C++ bitwise operations

- Memory is made up of bits and bytes.
- A bit is the smallest unit of storage in a computer, it stores a 0 or 1.
- A bytes consists of 8 bits – it is the smallest item that we can create a variable (char c);
- We can make use of every single bit by using bits wise operations:
  - Bitwise AND &: 0xff & 0x00 == 0
  - Bitwise OR |: 0xff | 0x00 == 0xff
  - Bitwise exclusive OR ^: 0xff ^ 0x00 == 0xff
  - Left shift <<
  - Right shift >>
  - Bitwise complement ~: ~0xf0 = 0x0f

Bitwise operation

- Unsigned char X = 8 (00001000)
- Unsigned Char Y = 192 (11000000)

- X = 00001000 00001000
- Y = 11000000 11000000
- X&Y = 00000000 X | Y = 11001000

- Test whether the second bit in X is 1?

  If (X & 0x04 != 0)

  - This allows logic based on bits in a variable.

Data structures

- Data structures help us write programs easier
  - Program = data structures + algorithms
- Data structures focus on organizing the data in a certain way such that operations on it can be efficient.
  - Picking the right way to store your data is very important for an efficient program.
- In Computer Science, many common data structures are used in many different applications – mastering data structures is essential for one to become a good programmer.
  - COP4530 systematically goes through most common data structures.
  - Most common data structures have been implemented in the C++ standard template library.
  - You are one of us AFTER you pass COP4530.

Data structure – an example

- For the word count program, what is the most important operations for the efficiency of the code?
  - Given a word, where is the word stored?
    - Search operation.
  - Your solution?
    - Array, or link list?
    - You have to go through the whole array to find the index that store the word. We call this an O(N) operation.
  - The ideal data structure for this assignment:
    - A hash table.
      - Give a word "abc", it is stored at Table[hashfunction("abc")].
      - The search is an O(1) operation.
  - For processing large files, using hashtable to replace the array can easily speedup the code by a factor of 100.

Data structure – another example

- Consider the problem of scheduling programs to run on CPU
  - This is a core function in operating systems. It happens very often (e.g. 50 times every second) and thus needs to be implemented in the most effective way.
- You have an array that stores the program information with priority.
  - You want to pick the highest priority program to run
  - The operation needs to be supported are
    - Insert a new program
    - Remove the program with the highest priority
  - How can we do both operations efficiently?
    - Array (sorted or not) is not efficient.
    - We have a data structure called priority queue that is ideal for this.

COP4530

- The class goes through many data structures
- You will understand how to use them and implement them
- Given a practical requirement, you will then know how to pick the right way to store the data to achieve efficiency.
  - This gives you a lot of tools to use in programming – with the tools, you will be able to write bigger and more efficient programs.

Abstract data types

- Data structures are usually implemented as abstract data types – stacks, Queues, Vectors, linked list, trees

- **Stacks**
  - First In Last Out (FILO). Insertions and removals from "top" position only
  - Analgoy - a stack of cafeteria trays. New trays placed on top. Trays picked up from the top.
  - A stack class will have two primary operations:
    - **push** -- adds an item onto the top of the stack
    - **pop** -- removes the top item from the stack
  - Typical application areas include compilers, operating systems, handling of program memory (nested function calls)

Queues

- First In First Out (FIFO). Insertions at the "end" of the queue, and removals from the "front" of the queue.
- Analogy - waiting in line for a ride at an amusement park. Get in line at the end. First come, first serve.
- A queue class will have two primary operations:
  - **enqueue** -- adds an item into the queue (i.e. at the back of the line)
  - **dequeue** -- removes an item from the queue (i.e. from the front of the line).
- Typical application areas include print job scheduling, operating systems (process scheduling).

Vector

- A data structure that stores items of the same type, and is based on storage in an array
- By encapsulating an array into a class (a vector class), we can
  - use dynamic allocation to allow the internal array to be flexible in size
  - handle boundary issues of the array (error checking for out-of-bounds indices).
- Advantages: Random access - i.e. quick locating of data if the index is known.
- Disadvantages: Inserts and Deletes are typically slow, since they may require shifting many elements to consecutive array slots

Linked list

- A collection of data items linked together with pointers, lined up "in a row". Typically a list of data of the same type, like an array, but storage is arranged differently.
- Made up of a collection of "nodes", which are created from a **self-referential** class (or struct).
  - **Self-referential class:** a class whose member data contains at least one pointer that points to an object of the same class type.
  - Each node contains a piece of data, and a pointer to the next node.
  - Nodes can be anywhere in memory (not restricted to consecutive slots, like in an array).
  - Nodes generally allocated dynamically, so a linked list can grow to any size, theoretically (within the boundaries of the program's memory).
- An alternative to array-based storage.
- Advantages: Inserts and Deletes are typically fast. Require only creation of a new node, and changing of a few pointers.
- Disadvantage: No random access. Possible to build indexing into a linked list class, but locating an element requires walking through the list.
- Notice that the advantages of the array (vector) are generally the disadvantages of the linked list, and vice versa

Tree

- A non-linear collection of data items, also linked together with pointers (like a linked list).
- Made up of self-referential nodes. In this case, each node may contain 2 or more pointers to other nodes.
- Typical example: a **binary tree**
  - Each node contains a data element, and two pointers, each of which points to another node.
  - Very useful for fast searching and sorting of data, assuming the data is to be kept in some kind of order.
  - **Binary search** - finds a path through the tree, starting at the "root", and each chosen path (left or right node) eliminates half of the stored values.