Review

- Linked list:
  - Doubly linked list, insertback, insertbefore
  - Remove
  - Search

Exception handling

C++ exception handling

- Exception: an error or problem condition
  - E.g. divide by zero, access NULL pointer, etc
- Exception handling: dealing with error or problem conditions
  - C++ has some built-in mechanism for exception handling
    - Without using the built-in support for exception handling, one can can such situations by adding checks in the code.
      - See sample1.cpp
      - Handling the exception: sample2.cpp in the traditional way

    - One can live without using C++ exception handling supports: just use the good old if-statement as shown in the example.

Why exception handling?

- Typical error-checking using the if-statement intermixes error handling code with the tasks of a program
- Many potential problems happen very infrequently.
  - Code to handle exceptions ideally should not intermix with the main program logic (making it hard to read and debug).

- With exception handling: code to handle exceptions is separated from the main program logic
  - improves program clarity.
- Exception handling also often improves program's fault tolerance – with a more systematic method to handle errors.

When to use exception handling?

- Exception handling is not always appropriate for handling exceptions
  - E.g. conventional method is better for input checking.

- When it is good?
  - Problems that occur infrequently.
  - Problems that do not need to be handled in the same block
  - Good for setting up uniform techniques for error-handling when many programmers work on multiple modules.

C++ exception handling

- The try-throw-catch blocks

```
try
{
   … code to try
   … possibly throw exceptions
}
catch (type1 catch_parameter_1)
{ … code to handle type1 exception}   // called exception handler
catch (type2 catch _parameter_2)
{ … code to handle type2 exception}
```

The try block

- Syntax:

```
Try
  {
     .. the main logic, possibly throw exceptions
  }
```

- Contains code when everything goes smoothly
  - The code however may have exceptions, so we want to "give it a try"
  - If something unusual happens, the way to indicate it is to throw an exception.

Throw statement

- Syntax:

  throw expression_for_value_to_be_thrown;

- Semantic:
  - This indicates that an exception has happens
  - If a throw statement is executed, the try block immediately ends
  - The program attempts to match the exception to one of the catch blocks (which contains code for exception handlers) based on the type of the value thrown (*expression_for_value_to_be_thrown)*
  - If a match is found, the code in the catch block executes
  - Only one catch block will be matched if any.
  - Program then resumes after the last catch block.

  - Both the exception value and the control flow are transferred (thrown) to the catch block (exception handler).

Catch block

- 1 or more catch blocks follow a try block.
- Each catch block has a single parameter with type
- each catch block is an exception handler (handling exceptions of one type)

```
catch (type catch_block_parameter)
{  … exception handler code
}
```

- Catch_block_parameter catches the value of the exception thrown, and can be used in the exception handler code.
- The exception thrown matches one of the catch parameter
  - If not, you have a un-caught exception situation.

Try-throw-catch summary

- Normally runs the try block then the code after the last catch block.
- If the try block throws an exception (run a throw statement)
  1. The try block stops immediately after the throw statement
  2. The code in the catch block starts executing with the throw value passed as the catch block parameter.
  3. After the catch block completes, the code after the catch blocks starts executing.
- See sample3.cpp
- If exception is thrown but not caught, then terminate() will be called – the function terminates the program. See sample3a.cpp

Multiple throws and catches

- Each catch block catches one type of exceptions
- Need multiple catch blocks
- When the value is not important, the parameter can be omitted.
  - E.g. catch(int) {…}
- catch (…) catches any exception, can serve as the default exception handler
- See sample4.cpp

Exception classes

- It is common to define classes just to handle exceptions.
- One can have a different class to deal with a different class of exceptions.
- Exception classes are just regular classes
  - They are just used for the exception handling purpose.
- See sample5.cpp

The C++ standard built-in exception class

- In C++, there is a standard library with pre-built exception classes. The primary base class is called exception, and comes from here:

```
#include <exception>
using std::exception;
```

- Your own exception class can be built from the standard exception class.
- See sample6.cpp

- All exceptions thrown from routines in C++ standard libraries.

Throwing an exception in a function

- So far, exceptions are thrown and caught in the same level in the samples.
- In practice, programs are modularized with routines
- Routines may be need to throw an exception that is catch in other routines
- See sample7.cpp

Throwing an exception in a function

- Function with potential to throw exceptions may behave in a strange manner.
- C++ allows specific potential exceptions for each routine using the exception specification.

```
double safedivide(int top, int bottom) throw (int);
```

- Can only throw the int type exceptions to the outside

```
double safedivide1(int top, int bottom);
```

- No exception list: the function can throw any exception.

```
double safedivide(int top, int bottom) throw ();
```

- Empty exception list: the function cannot throw any exception.