Exceptions and assertions

- so far, all our error detection and handling has been carried out explicitly by the programmer (us) inserting explicit checks
- many things can go wrong during execution:

  simple user input errors, failed memory allocation attempts,

  computations resulting in values to large to store,

  recursion to the point of running out of memory, etc etc
- some of these things can be regarded as part of normal processing (e.g. checking a user's input was within range/reprompting them)
- others might require us to interrupt the current program activity to solve the problem (e.g. dealing with a corrupted data structure)

Layers of handling

- most error checking of data is best handled at the “manual” level
  - e.g. checking for valid user input
- some unusual, but not entirely unexpected, circumstances can be best handled by leaving our current block of code and moving to a special handler of some form (***exceptions*** and ***exception handlers***)
  - e.g. when new fails to allocate the space we requested
- some circumstances we deem should never happen: they indicate a programming error, serious enough to terminate the program over
  - we can ***assert*** fundamental tests of we think ***MUST*** be true, requiring the program to *immediately* terminate if they are false

Exceptions: try and catch

- to indicate that we want to use exception handling for a block of code, we enclose it in a ***try*** block
- if we detect a situation that warrants it, we can ***throw*** an exception: this leaves the try block and searches for code to handle it
- following the try block we create ***catch*** blocks to capture and process the thrown exception

```
try {
    // normal code
    if (p == NULL) { // a made-up condition for now
       throw("Oh no, p is null!");
    }
   // more normal code
}
```

```
// would be right after the end of try
catch (std::string& emsg) {
    cout << "I caught " << emsg;
}

// then the rest of the program code
```

try/throw/catch doesn't return

- if a try block throws an error we leave the try block, go to the appropriate catch, and then go on with the rest of the program

  when the catch is done it doesn't come back to the try block

- if you want it to “try again” after a throw/catch you would need to enclose them in an outer loop, e.g.

```
do {
    try {
        // do stuff, possibly throwing someexcept
    }
    catch (someexcept e) {
        // deal with exception e
    }
} while (!done);
```

throw/catch parameters

- throw passes data parameters identifying the kind of exception and any data needed to process it
- a catch block's parameters specify what kinds of exception data it can process
  - we can have multiple catch blocks, each handling different kinds of exceptions, identified by their parameter lists
  - the exception is assigned to the first catch whose parameters match
  - we need to order the catch blocks from most specific (first) to most general (last), to ensure the right block catches an exception
  - the parameters can be any types, but usually we use classes (which can provide lots of data fields and supporting methods)
  - common practice: catch “by-reference”, and not as a const

The std::exception class(es)

- the <exception> library defines a widely used set of exception types/classes
- generally we use these where possible, and derive our own, more specialized, exception classes from them
- base class for exceptions is std::exception
- next level of classes derived from exception include: bad_alloc (for when new fails), bad_cast, bad_typeid, bad_exception, logic_failure, and runtime_error
  - exceptions derived from logic_error include domain_error, invalid_argument, length_error, out_of_range
  - exception derived from runtime_error include overflow_error, range_error, underflow_error

Example with new

- new throws a bad_alloc exception if it fails: catching this will handle more situations than a simple test for null after new

```
#include <iostream>
#include <exception>
#include <new>

int main() {
   long size = -1;
   int *ptr = NULL;
   try {
      ptr = new int[size];                    // throws exception cuz of the negative size
      std::cout << "would get here if nothing went wrong" << std::endl;
   }
   catch (std::bad_alloc& e) {
      std::cout << "new failed in try" << std::endl;
   }
}
```

Disabling exceptions

```
// we can sometimes disable throwing of exceptions,
// e.g. with new allocating objects
//        we can specify std::nothrow to have it return null instead of throwing an exception

SomeClass* array = new (std::nothrow) SomeClass[bigsize];

// if bigsize is too big this returns a null instead of throwing an exception
// (but still throws an exception if bigsize is negative!)
```

Adding new exception classes

- typically when creating our own exception classes we derive them from the most specific applicable descendant of std::exception
- if none of the classes seem applicable then we derive from std::exception itself, e.g.

```
class myexception: public std::exception { ...
```

throws/catches across functions

- a function's catch block(s) might not cover some exceptions it throws
- the exceptions can then be caught by the calling function
- we can thus have layers of try/catch blocks
- in the example on the next two slides:
  - main calls f2, f2 calls f1, f1 can throw three types of exception
  - we'll use a class heirarchy for the three types, based on std::exception
  - f1 catches one type itself, the most specific (class except1)
  - f2 catches a second type (class except0)
  - main catches the third, most general, type (class std::exception)
  - ***of course usually we'd pick more meaningful class names!***

Example: exception heirarchy

```
class except0: public std::exception {                  // deriving from the standard exception class
   protected:
      std::string message;
   public:
      // constructor expects just a string message as data
      except0(std::string msg): message(msg) { }
      virtual void print() { std::cout << “Type 0: “ << msg << std::endl; }
};

class except1: public except0 {
   protected:
      int info;
   public:
      // constructor expects both a string and an integer value as data
      except1(string msg, int val): except0(msg), info(val) { }
      virtual void print() override; {
          std::cout << “Type 1: “ << msg << “, “ << val << std::endl;
      }
};
```

Example: catches across functions

```
void f1(int opt) {
   try {
      std::exception E;
      if (opt == 0) throw except0("threw zero");
      else if (opt == 1) throw except1("threw one", 1);
      else if (opt == 2) throw E;
      std::cout << opt << " is ok!" << std::endl;
   }
   catch(except1& e) {
     e.print();
   }
}
void f2(int opt) {
   try {
      f1(opt);
   }
   catch(except0& e) {
     e.print();
   }
}
```

```
int main() {
    int data;
    try {
      std::cin >> data;
      f2(data);
    }
    catch(std::exception& e) {
        std::cout << "General exception!" << std::endl;
    }
}
```

* options 0-2 cause exceptions in f1, other values don't
* f1 can throw an exception constructor, or an exception object
* the catches specify which class they can handle

Cost/benefits of exceptions

- try/catch blocks allow us to separate code used to handle exceptions from the code dealing with regular program logic/control flow
  - pro: this makes the regular logic/flow cleaner and easier to read
  - pro: it can also allow centralization of the exception handler logic
  - con: it can become difficult to follow the exception trails
  - con: it does increase compile time and executable size
- most modern compilers & processors support efficient exception *checking*, but have high runtime performance cost if/when an exception does get thrown
  - for performance critical or memory-limited systems the cost might not be supportable, and retro-fitting existing code might not be feasible, but in most new “normal” programming projects the benefits outweight the performance costs

Assertions and assert

- the <*cassert*> library provides us with assert(condition), which immediately terminates the program if the condition is false
- typically used during development but turned off for the final product (when coding errors should have been eliminated)
- suppose at point X in a program we are certain that some pointer cannot possibly be null, if it ***is*** null then an actual coding error exists
  - terminate the program if the pointer is null, giving an error message specifying which assertion caused the termination

```
// code leading up to the check
assert(thepointer != NULL);
// rest of code
```