Exceptions & Templates

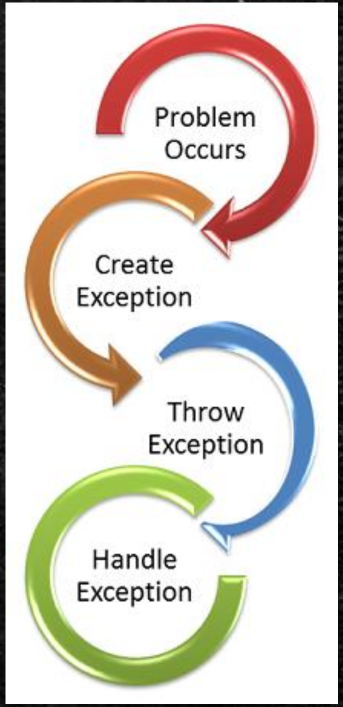
CS 150 – C++ Programming I
Lecture 14

Assumptions & Assertions Review

- Assumptions about valid inputs and outputs
  - Preconditions are inputs, postconditions are outputs
  - `@pre n should be >= 0 // sqrt precondition`
  - `@post status true if number read correctly`
- What to do about precondition violations?
  - Fix it silently, terminate with message, return error code, throw an exception, ignore it
- Use *assert* to automatically detect programming errors
  -
    ```
    int sum_between(int lower, int upper) {
      assert(lower <= upper); // cannot happen
    ```

Throwing Exceptions

- Errors caused by user input or by exceptional but anticipated circumstances
  - User types a filename incorrectly
  - Disk full when saving a file
- Should be handled by throwing an exception
  - `if (error condition) throw object;`
- Similar to a `return` statement from inside function
  - Does not return to calling function but to error handler
  - If no error handler, default handler terminates program



Types of Exception Objects

- In C++, any object may be thrown as an exception
  - In Java, only subclasses of *Throwable* may be thrown
  - `if (error) throw 42; // numbered codes`
  - `if (error) throw "OOPS"s; // C++ string`
  - `if (error) throw illegal_argument("a");`
- Standard library includes a variety of exception classes
  - `#include <stdexcept>`
  - `domain_error`: parameter outside the valid range
  - `invalid_argument`: invalid argument
  - `out_of_range`: argument not in its expected range

Handling Exceptions with *try* and *catch*

- To intercept and handle exceptions
  - Place the code which may fail inside a `try` block

  -
    ```
    try {
          int x = parseInt(str); // may fail
    }
    ```
  - Follow with any number of `catch` blocks
    - Specify type of exception to be caught (by reference)

  -
    ```
    catch (invalid_argument& e) {
      cerr << "Error: " << e.what();
    }
    ```
- Exercises: *try-catch* modifications

The *inthelper* Library

- Exercise: open *inthelper.h* and *inthelper.cpp*
  - Converts *string* to *int*
    - `int n = parseInt("42"); // returns 42`
  - Reads an *int* from the console
    - `int n = readInt("Enter a number: ");`
    - Prompt is optional
    - Keeps prompting until valid integer entered
- Document the functions in the header
  - Handle invalid input to *parseInt* by throwing an exception
  - Validate your logic with *assert*

Function Templates

```
int smaller(int a, int b)
{
  return a < b ? a : b;
}
```

- Consider this function:
  - Uses conditional operator to return smallest of a or b
- What if we want it to work for different types?

```
int main()
{
  auto a = smaller(3, 5);
  auto b = smaller(3.5, 7.5);
  auto c = smaller("zebra", "ant");
}
```

Option 1 - Overloading

- Write an overloaded function for each type

```
double smaller(double a, double b) {
  return a < b ? a : b;
}
string smaller(const string& a, const string& b) {
  return a < b ? a : b;
}
```

- Disadvantage? Have to write a new version for each type
  – Code in the body is exactly the same (redundant)

Option 2 – Write a Function Template

- Instructions to generate a function at compile time
  - Function only generated if called from your code
  - One template can generate many different functions

```
template <typename T> // or class
T smaller(const T& a, const T& b)
{
    return a < b ? a : b;
}
```

  - Generally placed in header file (not precompiled)

Two Ways to Call the Function

- Explicitly specify the type to be used for T
  - `auto s = smaller<string>("frog", "flea");`
- Implicitly allow the compiler to deduce the type
  - Calling this:
  - `auto n = smaller(3.46, 3.45);`
  - Generates this function:

  -
    ```
    double smaller(double a, double b)
    {
        return a < b ? a : b;
    }
    ```

Problems with Deduction

- Compiler can't read your mind!
  - `auto s = smaller("frog", "flea");`
  - Deduces type T as char array instead of *string*
- Solution? Add an explicit *string* overload of template
  -
    ```
    string smaller(const string& a, const string& b)
    {
       return a < b ? a : b;
    }
    ```

More Problems with Deduction

- What happens here?

  -
    ```
    auto n = smaller(3.46, 4);
    ```
  - Doesn't compile! Is T a double or an int?
- Add additional type parameters:

  -
    ```
    template <typename T, typename U>
    auto smaller(const T&  a, const U& b) {...}
    ```
  - Return type could be either T or U
  - Using auto with C++ 17 allows compiler deduction
  - In C++ 11/14 add a trailing return type instead (see Reader)
- Exercises: templates