1 Exceptions

1.1 Introduction

Exceptions are used to signify that a function is being used incorrectly. Once an exception is thrown, it is up to the programmer to decide how the exception is handled. Consider the following

```c++
int factorial(double n) {
  int m = 1;
  for (int i = 2 ; i <= n ; ++i) {
    m *= i;
  }
  return m;
}
```

This function calculates n!, which is only defined for integers \( n \geq 0 \). The question then arises: what should happen when the programmer attempts to call the factorial of a negative number or a non-integer? In these cases, we should throw an exception:

```c++
int factorial(double n) {
  if (static_cast<int>(n) != n) {
    throw std::invalid_argument("non-integer");
  } else if (n < 0) {
    throw std::invalid_argument("negative number");
  }
  int m = 1;
  for (int i = 2 ; i <= n ; ++i) {
    m *= i;
  }
  return m;
}
```

Above, we throw an std::invalid_argument object, passing in an error message to its constructor, when the user misuses the function. If one of the throw commands is reached in the code, the function will immediately unwind so that the code beginning at “int m = 1” will never be reached. In the main method above, we attempt to call the factorial function with the argument -5. We do not “handle” or “catch” the exception inside the main method meaning that we do not include a try-catch block giving instructions on how what the program should do when it sees the exception. As a result, the program exits immediately (really, the function std::terminate() is called), giving an error message along the lines of “terminating with uncaught exception of type std::invalid_argument: negative number”, and “Does not print” is never printed. To handle/catch the exception, we use a try-catch block:

```c++
int main() {
  bool again = true;
  if (again) {
    factorial(-5);
    std::cout << "Does not print" << std::endl;
  }
}
```
In this code, we query the user for a positive integer for which to calculate the factorial. If a positive integer is not entered, the factorial function throws an exception, and the code skips “again = false” and enters the catch block. Notice that the catch block will only except an object that can be bound to an std::invalid_argument const reference. Inside the catch block the “.what()” function of the exception object is called, printing out the error message used to construct the exception object inside of factorial. Here, the std::err object is used for printing out the error message. A dedicated error stream is used is because the buffer of std::cout may already contain some data, which we do not want to change just to print out an error message. Example output from the code above is given below

Example output:

```
−5
negative number
1.5
non−integer
5
120
```

If one does not know the specific details of a function, or if many possible exceptions can be thrown by a function and it is not desired to create many catch blocks, one can do the following

```c
int main() {
    try {
        function(input); // pseudocode
    } catch (const std::exception& e) {
        std::err << e.what() << std::endl;
    } catch (...) {
        std::err << "unknown exception caught" << std::endl;
    }
}
```

Here, a generic std::exception object is caught in the first catch block, which will catch many exceptions of the (exception) library. The elipsis in the second catch block will catch any exception not caught by the first catch block. Indeed, in the example above we only threw an std::exception object, but any expression can be thrown:
1.2 Exception safety and noexcept

The term “exception safety” refers to designing your code so that thrown exceptions do not cause memory leaks. Indeed, consider the following

```cpp
void whoops() {
    throw std::exception;
}

fun() {
    int* arr = int[1000];
    whoops();
    delete[] arr;
}

int main() {
    try {
        fun();
        // more code
    }
    // more code
}
```

The function fun creates a memory leak because fun exits before delete[] is used on the array. Thus, the code above is not exception safe. To make the code exception safe, one option is to use a try block inside of fun:

```cpp
fun() {
    int* arr = int[1000];
    try {
        whoops();
    }
    catch (...) {
        delete[] arr;
        throw;
    }
}
```

This can become annoying when writing code however. A more natural approach is to make sure that all dynamically allocated memory will be deleted when the stack is naturally unwound; i.e. we should remember the RAII idiom.

```cpp
void whoops() {
    throw std::exception;
}

fun() {
    std::unique_ptr<int[]> arr = std::make_unique<int[]>(1000);
    whoops();
}

int main() {
    try {
        fun();
        // more code
    }
    // more code
}
```

The code above is exception safe, because when the unique_ptr above is destroyed, the underlying data will be deallocated. Therefore, there will be no memory leak. (We have to trust that the unique_ptr class is exception safe!)

However, we don’t always have to be paranoid and imagine the worst-case scenario. The keyword noexcept can be used to guarantee that a function will not throw an exception:

```cpp
int fun() noexcept {
```
This code is exception safe because \texttt{fun()} will never throw an exception; thus, the \texttt{delete} statement will always be reached.

### 1.3 Stack unwinding

Stack unwinding refers to the process of an exception being passed up the call stack until a matching catch block is found to handle the exception. Consider the code below:

```cpp
void fun1() {
    throw -1;
}

void fun2() {
    fun1();
    std::cout << "1"" << std::endl;
}

void fun3() {
    try {
        fun2();
    } catch (const std::string& s) {
        std::cerr << "2"" << std::endl;
    } catch (const int x) {
        std::cerr << "3"" << std::endl;
        throw;
    }
}

int main() {
    try {
        fun3();
    } catch (...) {
        std::cerr << "4"" << std::endl;
    }
    std::cout << "5"" << std::endl;
}
```

We trace the order in which the code above runs:

1. \texttt{main} calls \texttt{fun3}, \texttt{fun3} calls \texttt{fun2}, \texttt{fun2} calls \texttt{fun1}.

2. \texttt{fun1} throws -1.

3. \texttt{fun2} does not handle the exception, so “1” is not printed and stack is unwound to \texttt{fun3}.

4. The exception is caught by the second catch block in \texttt{fun3}, printing out “3”. The exception is thrown again, and the stack is unwound to \texttt{main}.

5. In \texttt{main}, the general (...) catch block catches the exception, printing “4”. Finally, “5” is printed.
As the stack is unwound, data local to the scopes being unwound is cleaned up. For example,

```cpp
struct A {
    A(const int _x) : x(_x) {}
    int x;
    ~A() {std::cout << x << std::endl;}
}

void fun() {
    A a(1);
    throw -1;
}

void fun1() {
    A a(2);
    fun();
}

int main() {
    try {
        fun1();
    } catch (...) {
        std::cerr << 'hi' << std::endl;
    }
}
```

Here, we will see “1” print out, then “2.” When the exception is thrown in `fun()`, the variables local to `fun()` are deleted, causing the destructor of the variable “a” to be called. Similarly, in `fun1()`, the destructor of a is called as well.