Introduction to Exceptions and C++11 Threads

An **exception** is an abnormal condition that occurs during the execution of a program. For example, divisions by zero, accessing an invalid array index, or trying to convert a letter to a number are instances of exceptions. The concept is essentially the same in C++ as in Java, with a few syntax differences. Exceptions are used more commonly in Java than C++.

As an example, consider this toy example, which withdraws money from a pretend bank account. This version uses no exceptions. It just uses an if statement to check if we try to withdraw more money than exists in the account.

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
  int account = 100;  // Pretend we have $100 in our account
  int withdrawal;

  cout << "Enter an amount to withdraw." << endl;
  cin >> withdrawal;
  if (account - withdrawal < 0)
  {
        cout << "Not enough money!" << endl;
  }
  else
  {
        account -= withdrawal;
        cout << "You got " << withdrawal << " dollars." << endl;
        cout << "Your account has $: " << account;
  }
}
```

Nothing too exciting to see here – so let's go straight to an exception version of this program. In this case we can throw an exception when we try to withdraw more money than is in the account. We catch the exception and handle it somehow. In this case, we just print out the error message. The code is somewhat more complicated but normally we won't use exceptions quite this way. We can throw an exception of any data type and we can catch any data type, so if we wanted we could throw a string or double.

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
  int account = 100;  // Pretend we have $100 in our account
  int withdrawal;
```

```
  cout << "Enter an amount to withdraw." << endl;
  try
  {
   cin >> withdrawal;
   if (account - withdrawal < 0)
        throw 1;
   account -= withdrawal;
   cout << "You got " << withdrawal << " dollars." << endl;
   cout << "Your account has $: " << account;
  }
  catch (int e)
  {
        cout << "Exception code is " << e << " withdrawal problem"
             << endl;
  }
}
```

The advantage of this approach is we can now throw multiple exception codes for different types of exceptions. For example, we could throw the string "Withdrawal exceeds max" if we want to have an exception for trying to withdraw over 1000:

```
int main()
{
  int account = 100;  // Pretend we have $100 in our account
  int withdrawal;

  cout << "Enter an amount to withdraw." << endl;
  try
  {
   cin >> withdrawal;
   if (withdrawal > 1000)
        throw "Withdrawal exceeds max";
   if (account - withdrawal < 0)
        throw 1;
   account -= withdrawal;
   cout << "You got " << withdrawal << " dollars." << endl;
   cout << "Your account has $: " << account;
  }
  catch (int e)
  {
        cout << "Exception code is " << e << " withdrawal problem"
             << endl;
  }
  catch (char const *s)
  {
        cout << s << endl;
  }
}
```

I used "char const *s" as the type for the second catch, since throwing " " by default in C++ is a c-style string, not a STL string.

More typically, the reason for the try-throw-catch is because the code that detects the exception doesn't know how to handle it. In the above examples, it is probably better written using a regular if-statement because we know how to handle the different situations that might arise.

When wouldn't we know how to handle it? Consider a function that does the withdrawal calculation. Maybe this calculation is done on the bank server or somewhere that doesn't have access to the console to print error messages to the user. The typical solution is to return some error code:

```
int withdraw(int account, int withdrawal)
{
   if (account - withdrawal < 0)
   {
        // What do we do if we can't use output to the user here?
        // Maybe return an error code
        return -1;
   }
   // Return new balance
   return account - withdrawal;
}

int main()
{
  int account = 100;  // Pretend we have $100 in our account
  int withdrawal;

  cout << "Enter an amount to withdraw." << endl;
  cin >> withdrawal;
  account = withdraw(account, withdrawal);
  if (account > -1)
  {
   cout << "You got " << withdrawal << " dollars." << endl;
   cout << "Your account has $: " << account;
  }
  else
    cout << "Withdrawal error." << endl;
}
```

This works (there are perhaps better ways to do this, like passing account by reference) but what if we want to allow negative account balances? Then there isn't a way to distinguish the return value as an error code from the actual account balance. It also relies on the user checking the error code.

An alternate solution is for the function to throw an exception and abort. Essentially the function is saying "I don't know how to handle this issue, but it happened, so whoever called me, you get to handle it." To do this, we normally define our own exception class. It can be trivial. Here are two, one for exceeding the maximum withdrawal amount, and the other for exceeding our balance:

```
class ExceedsMax
{ };
class ExceedsBalance
{
 public:
   ExceedsBalance() { message = "Need mo money.";};
   ExceedsBalance(string s) { message=s; };
   string message;
};
```

Here is how we modify the withdraw function to throw these exceptions. We have to tack on the exceptions thrown at the end of the function header:

```
int withdraw(int account, int withdrawal)
            throw (ExceedsMax,ExceedsBalance)
{
   if (withdrawal > 1000)
        throw ExceedsMax();
   if (account - withdrawal < 0)
        throw ExceedsBalance("Not enough money in the account.");
   // Return new balance
   return account - withdrawal;
}
```

If we leave main unchanged, everything is happy if there is no exception:

```
Enter an amount to withdraw.
30
You got 30 dollars.
```

But if we enter an exception condition:

```
Enter an amount to withdraw.
102
terminate called after throwing an instance of 'ExceedsBalance'
```

The program crashes! If we don't catch the exception thrown then C++ "catches" it for us, which basically makes the program stop. In this way, throwing exceptions is like the nuclear option. We could return error codes that the programmer might ignore. But the programmer can't ignore an exception (well, except by writing an empty catch block). To keep the program from crashing we would handle these exceptions in main:

```
int main()
{
  int account = 100;  // Pretend we have $100 in our account
  int withdrawal;

  cout << "Enter an amount to withdraw." << endl;
  cin >> withdrawal;
  try
  {
```

```
    account = withdraw(account, withdrawal);
    cout << "You got " << withdrawal << " dollars." << endl;
    cout << "Your account has $: " << account;
  }
  catch (ExceedsBalance e)
  {
    cout << e.message << endl;
  }
  catch (ExceedsMax e)
  {
    cout << "You exceeded the maximum withdrawal amount." << endl;
  }
}
```

Sometimes exceptions are overused – once again they are most appropriate inside a function that isn't able to handle the exception case itself. In that case, throw an exception and let the calling code handle it.

C++11 Threads

In C++11 a wrapper Thread class is available. However, you still need to have a library that implements the thread class. Visual Studio 2015 supports C++11 threads, but earlier versions don't. In Linux we will need to use the pthread library.

A thread is a separate computation process. In Java and C++, you can have programs with multiple threads. You can think of the threads as computations that execute in parallel. On a computer with enough processors, the threads might indeed execute in parallel. In most normal computing situations, the threads do not really execute in parallel. Instead, the computer switches resources between threads so that each thread in turn does a little bit of computing. To the user this looks like the processes are executing in parallel.

You have already experienced threads. Modern operating systems allow you to run more than one program at the same time. For example, rather than waiting for your virus scanning program to finish its computation, you can go on to, say, read your e-mail while the virus scanning program is still executing. The operating system is using threads to make this happen. There may or may not be some work being done in parallel depending on your computer and operating system. Most likely the two computation threads are simply sharing computer resources so that they take turns using the computer's resources. When reading your e-mail, you may or may not notice that response is slower because resources are being shared with the virus scanning program. Your e-mail reading program is indeed slowed down, but since humans are so much slower than computers, any apparent slowdown is likely to be unnoticed.

Threads are useful when you need extra speed and want to run computations (likely) in parallel, and also when you want some processing to continue when another part is blocked/stopped (perhaps waiting for input). In GPU programming you can have hundreds of thousands of threads! It is possible to run what used to be the equivalent of a supercomputer on a GPU-enabled server or workstation.

Let's go straight to some code to show how to fire off a function that runs in a separate thread:

```
#include <iostream>
#include <thread>

using namespace std;

void func(int a)
{
    cout << "Hello World: " << a << " " << this_thread::get_id()
<< endl;
}

int main()
{
    thread t1(func, 10);
```

```
    thread t2(func, 20);
    t1.join();
    t2.join();

    return 0;
}
```

Compile this program in Linux with:

g++ program.cpp –std=c++11 -lpthread

This program starts off two threads, each runs the function "func". Each thread is automatically given a unique ID, which we can access if desired from get_id(). It is often useful to pass in an ID number, which we did by passing in the number in variable a.

If you run this, you'll see the two threads run and output Hello World. Once a thread is started we have no control over when it runs – it is now up to the operating system! You can see this by running the threads and seeing the output from each thread overwriting the other. This is because while one thread is in the middle of outputting its message, there is a context switch and we run the second thread, which spits out its text right in the middle of the text from the first thread.

The join() function makes the main function wait for each thread to finish before continuing. This is important to synchronize multiple threads.

If we want to avoid the threads overwriting each other, we can add a **mutex**, for mutual exclusion. This locks the thread so only one thread can enter a region of code at a time. This is extremely important for some programs to prevent deadlock or other types of errors (you see more of this in operating systems). The following modification forces other threads to wait so only one at a time can run the code in func:

```
#include <mutex>

using namespace std;

mutex global_lock;

void func(int a)
{
    global_lock.lock();
    cout << "Hello World: " << a << " " << this_thread::get_id()
<< endl;
    global_lock.unlock();
}
```

It is common to want a lot more than one or two threads. In this case, we can make an array of threads. Here is some code in main that makes an array of 10 threads:

```
thread tarr[10];
for (int i =0; i < 10; i++)
    tarr[i] = thread(func, i);
for (int i =0; i < 10; i++)
```

```
        tarr[i].join();
```

Notice the unpredictability of which thread runs first!

```
      Hello World: 0 140198342674176
      Hello World: 3 140198311204608
      Hello World: 2 140198321694464
      Hello World: 4 140198300714752
      Hello World: 1 140198332184320
      Hello World: 5 140198290224896
      Hello World: 6 140198279735040
      Hello World: 7 140198269245184
      Hello World: 8 140198258755328
      Hello World: 9 140198248265472
```

It is common to want to run a class in a thread. Here is the template. In this case we called the class "Runnable" but it could be whatever name you like:

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

class Runnable
{
  public:
    Runnable();
    Runnable(int n);
    void operator()();  // Yes, there are two ()()
  private:
    int num;
};
Runnable::Runnable() : num(0)
{ }
Runnable::Runnable(int n) : num(n)
{ }
void Runnable::operator()()
{
        cout << "Hello world, I am number " << num <<  endl;
}

int main()
{
    Runnable r1(10);
    Runnable r2(20);

    thread t1(r1);
    thread t2(r2);

    t1.join();
    t2.join();
    return 0;
}
```

When the thread starts, the class Runnable executes the code in the operator()() method. Any data we want to pass to the thread is generally sent in the constructor.

Here is one more example. In this case, we do something useful! We have three threads, and each one is searching (perhaps in parallel) a portion of an array for the minimum value. The minimum each thread finds for each section is stored in a "results" array, where we have a slot reserved for each thread. The main function has to go through the results to find the overall minimum.

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

class Runnable
{
 public:
  Runnable();
  Runnable(int *target, int *results, int num, int start, int end);
  void operator()();
 private:
  int *target, *results;
  int num, start, end;
};

Runnable::Runnable()
{
 target=nullptr;
 results=nullptr;
 num=0;
 start=0;
 end=0;
}
Runnable::Runnable(int *target, int *results, int num, int start,
                   int end)
{
  this->target= target;
  this->results = results;
  this->num = num;
  this->start = start;
  this->end = end;
}
void Runnable::operator()()
{
 int min = target[start];
 for (int i=start+1; i<=end; i++)
 {
        if (target[i]<min)
                min = target[i];
 }
 results[num] = min;
}
int main(int, char **)
{
```

```
    thread tarr[3];
    int target[] = {31, 66, 41, 8, 92, 47, 22, 87, 45, 92, 4, 14};
    int results[] = {999, 999, 999, 999};
    for (int i = 0; i < 3; i++)
    {
         Runnable r(target, results, i, i*4, i*4+3);
         tarr[i] = thread(r);
    }
     for (int i =0; i < 3; i++)
         tarr[i].join();
     for (int i =0; i < 3; i++)
         cout << results[i] << endl;
     int min = results[0];
     if (min > results[1])
         min = results[1];
     if (min > results[2])
         min = results[2];
     cout << "The minimum from threaded min-search is " << min << endl;
     return 0;
}
```

The output:

```
8
22
4
The minimum from threaded min-search is 4
```

The code sends in the array to search, an array for results, an ID, and bounds for each thread to search. Each thread then searches its portion of the array, finds the minimum, and uses its ID to determine a unique spot to place its result in the results array.

There is a lot more to threads but at least you should have an idea of what they are and where they may be useful from this brief introduction.