Some Other Topics: Exceptions and Static Fields

Exceptions

Things do not always go as planned in programs:

- Files need not exist.
- Memory can be full.
- Disk space can be full.
- Square roots, logarithms need not exist.
- User input can be incorrect.

A good program should try to do something meaningful when something bad happens. (Print error message, ask for new input, try to reserve disk space again at a later moment)

Exceptions (2)

A program can be made robust by inserting many if statements:

```
int dosomecomputation( int x )
{
   int* p = new int [ (some big number) ];

   if( !p )
   {
      // There is a problem, we cannot continue
      // the computation.

      std::cerr << "We ran out of memory. ";
      return  // what should we return? How do we check
              // in the calling function that something
              // went wrong?
   }
```

```
}
```

If the memory could not be allocated, every function that called **dosomecomputation** (a complete stack) has a problem, and needs to make some decision what to next.

Exceptions (3)

There are many possible other solutions, but none is really good:

- Define a a partial<X> class. It contains either an X, or an error message.
- Define a union<X1,X2> class.
- Use additional parameters of type std::string& error, to which you can assign an error.

In all cases, you have to write if statements in every function that might call the problematic function, and you have a problem if you forget one somewhere.

Exceptions (4)

It would be much nicer if there would be some automatic way of travelling down on the call stack until you reach a point where you know what to do.

This can be done automatically by throwing an exception.

```
throw n;
```

In principle, everything can be thrown, but one should throw only objects that inherit from one of the standard exceptions in the library.

Exceptions (5)

If you know what to do with an exception, you can write

```
try
{
   (code in which a throw statement can occur)
}
catch( N1& n1 ) { do something }
catch( N2& n2 ) { do something }
```

throw/catch is a fundamental control construct, like while, if, do.

Exceptions (6)

The standard library defines exceptions of two types, `std::logic_error` and `std::runtime_error`.

The C^{++} standard book says that **logic_error** are exceptions that could be caught at compile time by analyzing the program. **run_time** error are the other errors.

I think that this distinction makes no sense.

If you are writing a library, or part of a large project, then possibly an error is made inside the project but not by an end user.

This person is still a user of your library. At his compile time, the error can be caught, but not at your compile time.

Exceptions (7)

When an exception is thrown, destructors of all variables are called, until the exception is caught.

Make sure that all resources (files, memory, locks), are held by class objects that have a destructor that returns them. In this way, leaks are impossible.

RAII : Resource Acquisition is Initialization.

assert

`assert(b);`

Check if condition `b` holds, and abort the program if `b` is false. It can be used for checking preconditions.

```
double operator[] ( size_t i ) const
{
   assert( i < size( ));
   return .. something.
}
```

Don't Use assert, because quitting is only acceptable in toy programs. Throw an exception instead. Somebody who uses your code, can decide to catch it.

Static Members

The key word static can be used in the following three ways:

1. For fields of classes. In that case, the field is a single variable that exists independent of the class.

2. For member functions of classes. In that cases, the member function can be called without class element.

   A static member function can only call other static member functions of the class. It can also access static fields.

3. A static local variable is created when the function is called for the first time, and exists ever after.

   When the function recursively calls itself, the same local variable is used.

Static Field Initialization

Initialization of static fields in C^{++} is problematic. (It is called static initialization order fiasco)

```
(File A.h)

class A
{
   static int a;
}

(File A.cpp)

int A::a = 44;
```

```
(File B.h)

class B
{
   static int b;
}

(File B.cpp)

int B::b = A::a + 1;
```

Is this going to work? Nobody knows.

The only thing that one can know is:

Static initializations occurring in the same file are done in the order in which they appear in the file.

Avoiding the Fiasco

The simplest solution is to replace the static variables by functions:

```
(File A.h)

class A
{
   static int& a( );
}

(File A.cpp)

int& A::a( )
{
   static x = 44;
   return x;
}
```

(File B.h)

```
class B
{
   static int& b( );
}
```

(File B.cpp)

```
int& B::b( )
{
   static x = a( ) + 1;
   return x;
}
```

Probably better is to completely avoid static class variables. unless for simple things like counting how often a function is called.

Invisible data streams are always a risk.

Just make a local variable and pass it explicitly.

Static member functions are fine. Static global (outside of class) functions should not be used.