Some Other Topics: Exceptions, keywords `static` and `constexpr`

Exceptions

Things do not always go as planned in programs:

- Files need not exist.
- Memory can be full.
- Disk space can be full.
- Sending/receiving a message may fail.

A good program should try to do something meaningful when something bad happens. Terminating is only possible in toy programs.

Exceptions (2)

A program can be made robust by inserting if statements:

```
int dosomecomputation( int x )
{
   int* p = new int [ (some big number) ];
   if( !p )
   {
      // We ran out of memory.

      return  // what should we return? How do we check
              // in the calling function that something
              // went wrong?
   }
   do the computation; delete[] p; return result;
}
```

Exceptions (3)

How to check validity of result?

- Use a special value? (Not always possible.)
- Use boolean parameter, e.g.

  `std::pair<bool,int> dosomecomputation( int x )`, or

  `dosomecomputation( int x, bool& valid )`

  May require default constructor.

Even if solution is found, every function that calls **dosomecomputation** must check validity.

Programmers are lazy, repetitive code is bad, checking costs time, functions should have nice interfaces.

Exceptions (4)

Exceptions create an alternative way of exiting from a function by **throwing** an exception.

The alternative way of exiting is continued, until the exception is **caught**.

- Between point of throwing, and point of catching, no checks are necessary.
- No cost when exception is not thrown, no corruption of interface, no repeated code.

Exceptions (5)

Exception is thrown with `throw expression`.

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

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

If you don't know what to do, don't catch.

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

Exceptions (6)

```
int dosomecomputation( int x )
{ int* p = new int [ (some big number) ];
      // new throws std::bad_alloc( ), when it cannot
      // get the memory.
   do the computation; delete[] p; return result;
}


try
{ int d = dosomecomputation( 44 );
   std::cout << "result of computation = " << d << "\n";
}
catch ( std::bad_alloc& al )
{
   std::cout << "not enough memory to complete computation\n";
}
```

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

The C^{++} standard book says that one should only throw exceptions that inherit from `std::logic_error` and `std::runtime_error`. `logic_error` are exceptions that could be caught at compile time by analyzing the program. `runtime_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.

Main thing is that you only exceptions that inherit from either of the exceptions above.

RAII

When an exception is thrown, destructors are automatically 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 the resource. In this way, leaks are impossible.

RAII : Resource Acquisition is Initialization.

RAII (2)

dosomecomputation( ) is not a good implementation, because `p` will not be deleted, if 'do the computation' throws an exception.

Either use `std::vector<int>`, or create

```
struct guard
{
    int* p;
    guard( int *p ) p{p} { }
    ~guard( ) { delete[] p; }
};

guard guarded_p = new int[ ... ];
   // 'Forgetting' to delete is impossible.
```

Exception thrown from Constructor

It depends from where:

```
struct myclass
{  A a;
   B b;
   C c;

   myclass( const A& a, const B& b, const C& c )
      : A{a},
        B{b},  // If throws, do: a. ~A( );
        C{c}   // If throws, do: b. ~B( );  a. ~A( );
   {
      If throws, do: c. ~C( ); b. ~B( ); a. ~A( );
      Note that this -> ~myclass( ) is not called.
   };
};
```

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 objects. It exists through the life time of the program.

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

   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 Fields of Classes

```
class A
{
   static int i;
}

A::i = 44;
   // Pretty much like namespace.

(somewhere else)
std::cout << A::i << "\n";
   // prints 44.
```

Static Member Functions in Classes

```
class A
{
   int f;
   double g;
   static int x;

   static int gcd( int x1, int x2 );
      // Cannot use fields f and g, could use x.
};

std::cout << A::gcd( 3, 4 ) << "\n";
   // Can be called without creating an A.
```

Static Local Variables in Functions

```
int somefunction( int x )
{
   static unsigned int counter = 0;
      // Initialization '=0' done only once.

   std::cout << "this is the " << ++counter << "-th call\n";
      // Printed every time.
}
```

Incorrect Uses

In *C*, `static` keyword was used to make a name local to a file, (as opposite to `extern`).

I write this, because you may see it in old (bad) code. Don't write this by yourself.

If you want local names, create a namespace, or an anonymous namespace.

```
namespace {
   int a;
   int b;
}

a,b visible here, nowhere else.
```

Constexpr

Purpose of `constexpr` is to make the preprocessor unemployed:

```
int p[ 100 ];    // OK.

// With preprocessor:

#define BIGENOUGH 100
int p[ BIGENOUGH ];
   // Scoping rules are unclear. Ignores
   // namespaces.

const static int bigenough = 100;
int p[ bigenough ];
   // Should not compile (but does with GCC).
```

Constexpr (2)

```
constexpr static int bigenough = 100;
int p[ bigenough ];
```

Use `constexpr` to mark functions and constants that can be computed at compile time:

```
constexpr int i = 4;
constexpr square( int i ) { return i*i; }

int p[ square(i) ];
   // Allowed.
```

The rules for `constexpr` functions are very restrictive, but changing. Look at the standard for C^{++}-17.