

A Whirlwind Tour of C++

Robert P. Goddard
Applied Physics Laboratory
University of Washington

Part 1: 3 November 2000: Object Model

Part 2: 17 November 2000: Templates

Part 3: 15 December 2000: STL

Part 4: 30 March 2001

Exceptions and Exception Safety

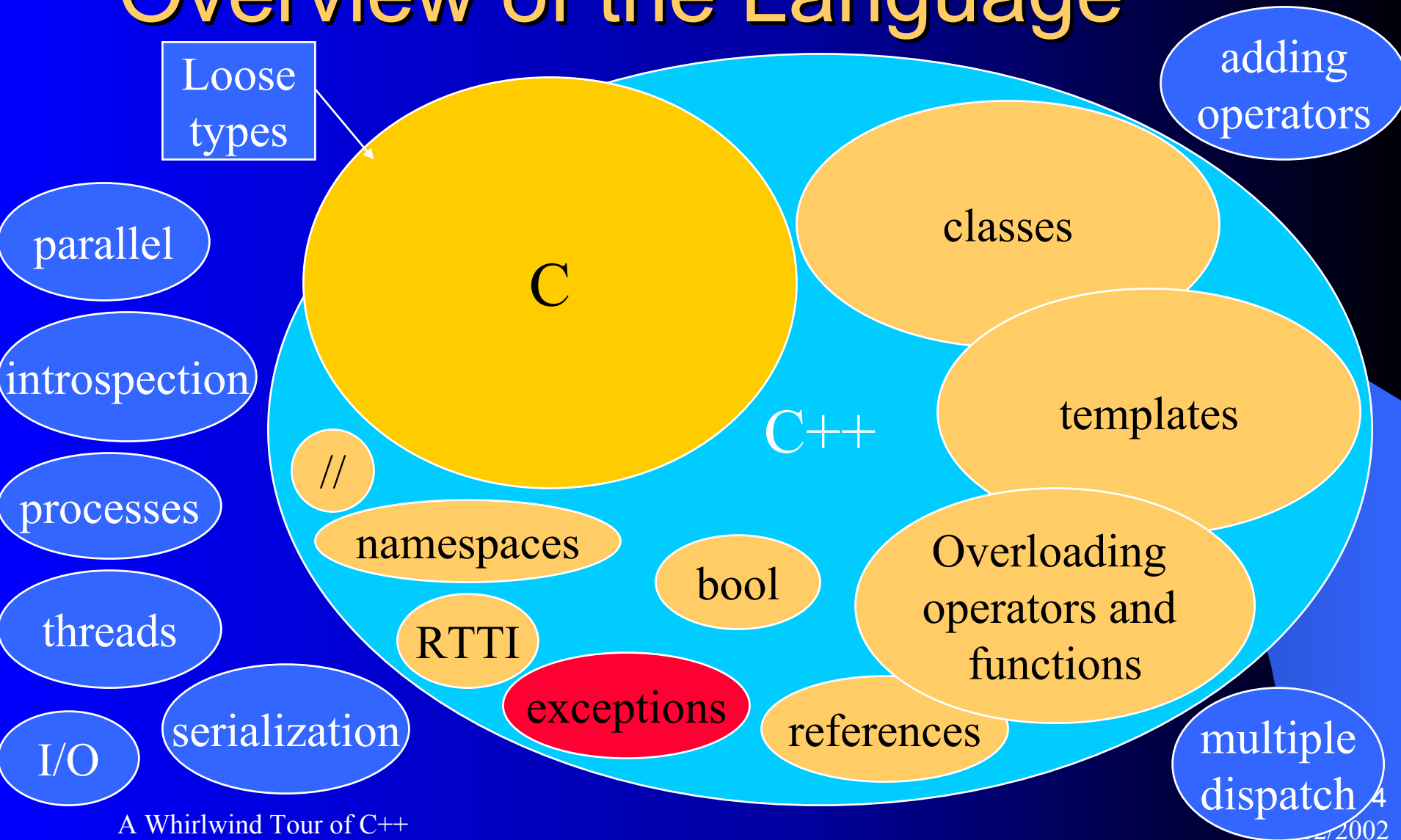
Introduction

- Scope: ISO/ANSI Standard C++ language and standard library
- Audience: Programmers. Some familiarity with basic C++ object model will help.
- Fast and shallow. It's simple on the surface, but the implications are complicated.

Series Agenda

- Overview of language features
- Overview of standard library
- Objectives of the language
- Classes and objects
- Templates
- Standard Template Library
- **Exceptions**
- I/O Streams
- Numerics
- etc.

Overview of the Language



Ways to Indicate Failure

- **Set a global error flag** like C math library. Not thread safe, universally ignored.
- **Return a status flag** like C stdio. Almost always ignored, OR error handling obscures normal logic. Useless for constructors, operators.
- **Set flag in object** like old iostreams. Usually ignored by client, slows member functions.
- **Print a message and exit().** Draconian.
- **Use assert().** Draconian, removed by NDEBUG.
- **Call longjmp().** Nonlocal go-to. Obscure, invisible, leaky (no destructor calls).
- **Throw an exception.** Imperfect but mandatory.

Throwing an Exception

```
#include "Complex.h"
#include <stdexcept>
class Complex::DivZero : public std::exception {
    const char *what() const // override
        { return "Complex divide by zero"; }
};
```

```
Complex& Complex::operator/=(
    const Complex& den ) throw (Complex::DivZero)
{ if ( den == Complex(0.0,0.0) )
    throw ComplexDivZero();
  // ... do the calculation ...
  return *this;
}
```

Catching an Exception

```
#include <stdexcept>
void MyFun() { // no exception specification
    try {
        // ... calculation involving Complex etc.
    }
    catch (Complex::DivZero& d) {
        // ... recover and carry on
    }
    catch (std::exception& e) { // base class
        cerr << "Caught: " << e.what() << endl;
        throw; // re-throw same exception
    }
    catch (...) { // anything else
        throw MyOwnException(); // different
    }
}
```

Exceptions Call Destructors for Local (Automatic) Variables

```
TypeB FunA() { // No exception spec: Can throw any
    TypeA a; // more code
    if ( goof ) throw TypeX();
    return TypeB( a );    // Skipped on throw
    // a destructor always called here
}
```

```
void FunB() {
    TypeB b;
    TypeB bb = FunA();    //On throw, no bb
    // Destructor: bb if it exists
    // Destructor: b always
    // No catch: Exception propagates up
}
```

Exceptions Call Destructors 2

```
void FunC() {
    TypeA a;
    try {
        TypeC c;
        FunB();    // Might throw
        // c always destroyed here
    }
    catch (TypeX& x) {
        //... recover
    }    // Carry on
        // a destroyed here
}
```

Exceptions In Constructors

```
class TypeA {
    Type1 member1; Type2 member2; Type3 *pmember3;
public:    TypeA();    // constructor
}
TypeA::TypeA()
: member1(),    // may be omitted
  member2( value2 ), pmember3( 0 )
{
    pmember3 = new Type3( value3 );
}
```

- If anything throws, previously constructed members are destroyed in reverse order. The TypeA object does not exist, never did exist, hence will not be destroyed.
- This example is exception safe (if destructor deletes #3) and exception neutral (propagates exceptions to caller).

Summary (surface)

- Throw at any level unwinds call stack until a matching catch is found. At each level, destructors for local variables are called before leaving function. No leaks (in principle).
- Any *copyable* type may be thrown & caught. Normally throw a special-purpose class, often based on `std::exception` hierarchy. Copied to special area (not on stack).
- Throw where error is discovered. Catch where you have enough context to recover.
- Throw by value, catch by reference.
- No time penalty unless exception is thrown.

Standard Exceptions

Name	Thrown by	Header
<i>bad_alloc</i>	<i>new</i>	<i><new></i>
<i>bad_cast</i>	<i>dynamic_cast</i>	<i><typeinfo></i>
<i>bad_typeid</i>	<i>typeid</i>	<i><typeinfo></i>
<i>bad_exception</i>	<i>exception specification</i>	<i><exception></i>
<i>out_of_range</i>	<i>at()</i> (containers), <i>bitset<>::operator[]()</i>	<i><stdexcept></i>
<i>invalid_argument</i>	<i>bitset constructor</i>	<i><stdexcept></i>
<i>overflow_error</i>	<i>bitset<>::to_ulong()</i>	<i><stdexcept></i>
<i>ios_base::failure</i>	<i>ios_base::clear()</i>	<i><stdexcept></i>

Exception Safety: 3 Levels

- **Basic Guarantee:** Operation doesn't leak resources. State remains consistent, but not necessarily predictable. Fairly easy.
- **Strong Guarantee:** Operation either succeeds or leaves state unchanged. Commit-Rollback semantics. Depends critically on class design.
- **Nothrow Guarantee:** Operation will never emit an exception. Overall safety is impossible unless *some* operations don't throw.

Basic Canonical Safety Rule

- Never allow an exception to escape from a destructor or from overloaded `operator delete()` or `operator delete[]()`.
- If destructor throws while another exception is uncaught, `std::terminate()` is called. The end.
- Fortunately, library destructors and deletes don't throw.

Leak Prevention 1: Try Blocks

```
void Fun1() {  
    Type1 *p1 = new Type1();  
    try {  
        // do some stuff  
    } catch ( ... ) {  
        delete p1;  
        throw;  
    }  
    delete p1;  
}
```

- Verbose, repetitive, error prone.
- Multiple resources require nested try blocks.

Leak Prevention 2: `auto_ptr`

```
#include <memory>
void Fun1() {
    std::auto_ptr<Type1> p1( new
        Type1() );
    // do some stuff
}
```

- `auto_ptr` destructor destroys and deletes its referenced object, whether or not an exception is thrown!
- Single objects only, not arrays. But...

Leak Prevention 3: STL Containers

```
#include <vector>
void Fun1(int size) {
    std::vector<Type1> v1(size);
    // Use v1 just like an array
}
```

- **std::vector** destructor destroys and deletes its contents, whether or not an exception is thrown!

Leak Prevention 4: Constructors

```
class TypeA {  
    std::vector<Type1> vec1;  
    std::auto_ptr<Type2> ptr2;  
public:  
    TypeA(int size)  
        :   vec1( size ),  
           ptr2( new Type2() ) {}  
};
```

- This version is exception safe.
- No destructor is required! The one the system invents automatically does the right thing!

Resource Acquisition Principle

- Acquire resources in constructors.
- Release resources in destructors.
- Applies to memory, files, mutexes, locks, devices, ...
- If necessary, invent a tiny class whose only function is to acquire and release a resource (in constructor and destructor).
- The C++ library can often make it automatic.

Leak Prevention 5: swap()

```
doit( std::vector<TypeA>& v )
{
    std::vector<TypeA> w( v.size() );
    // Compute in w (may throw)
    v.swap( w );    // Doesn't throw
}
```

- This one satisfies strong guarantee: Success or no change in state.
- All STL containers and `auto_ptr` have `swap()`.
- Consider adding `swap()` to your classes.

Leak Prevention 6: pimpl idiom

```
class TypeA {  
    class Impl { // the guts }  
    std::auto_ptr<Impl> pimpl;  
public: // constructor etc...  
}
```

- To satisfy strong guarantee: State changing operations construct new, temporary `TypeA::Impl`, do operation, then swap if successful. Temporary is destroyed on exit.

Where to Get More Information

- Book list posted at:

<http://staff.washington.edu/rpg3/booklist.html>

- Bjarne Stroustrup, *The C++ Programming Language*, Special (or 3rd) Edition, Addison-Wesley, 2000
- Herb Sutter, *Exceptional C++, 47 Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, 2000