C++ Internals
Tighter Code Through Understanding

Tyson Whitehead

SHARCNET
The University of Western Ontario

June 1, 2009

Name Mangling

The compiler accomplishes various things through the use of name mangling. That is, it adds stuff into the name of your functions behind your back in order to distinguish them. This is mostly useful.

Caveat Emptor

- What about compilers that use different encoding schemes?

Name Mangling — Function Overloading

The compiler encodes into your function names is the functions arguments. This adds the ability to create functions of the apparent same name with different arguments.

Caveat Emptor

- What about ambiguous situations?

Name Mangling — Namespaces

The compiler encodes an optional namespace into function and structure/class names. This avoid name collisions between modules and is quite useful.

Name Mangling — External Functions

In order to use a function compiled without name mangling (e.g., a C function), the compiler has to be told to disable name mangling for the function.

Classes and Structures

It is convenient to group data structures together with the functions that operate on them in one place. This is useful and greatly increases maintainability. It is also nice to have some syntactic sugar to extend structures and handle function tables.

Caveat Emptor

- Claimed benefits beyond this are largely overrated.

The compiler converts operator expressions into series of functions calls to functions named operator+, operator−, etc. Overloading these functions makes it possible add support for new structures/classes.

Caveat Emptor

- Operator overloading frequently leads to very poor performing code due to having to generating a lot of temporaries.

Classes and Structures — Inheritance

Structures/classes can be extended by deriving further classes. Internally, the compiler creates a new structure/class with the first members being the structures/classes being derived from.

Virtual functions allow functions in base structures/classes to be overridden. This is done by creating tables of function pointers for the various combinations of virtual functions and creating a pointer in the underlying data structure to point to the appropriate table. Virtual functions calls are then routed through the table.

Caveat Emptors

- The indirections adds a performance hit (and restricts inlining).
- The functions which can be overridden have to all be specified ahead of time (invariably the one you want is not).
- The compiler cannot assume it knows everything about the function (an exception handling performance hit).
- Non-virtual destructors can (will) lead to grief.

Making derived classes by just making the first members of the underlying data structure the structures/classes they are derived from results in multiple copies of any structure/class that occurs multiple times in object hierarchy. Making a base classes virtual forces all accesses to go through a pointer. Multiple copies then becomes multiple pointer. This solves the problem as the latter can all be made to point to a single copy of the underlying data.

Caveat Emptors

- The indirection adds a performance hit.
- The bases which might occur multiple times are all specified ahead of time (invariably the one you want is not).

Exceptions

The compiler maintains a stack of handlers associated with various stack frames to jump to when an error is raised. Local classes/structures are automatically wrapped in try catch blocks in order to ensure their proper destruction.

Caveat Emptors

- Maintaining information necessary to handle exceptions adds a lot of overhead, and, unless the compiler knows for sure that no exceptions will happen, it has to add this overhead in.
- A function call cannot be guaranteed to throw no exceptions unless the programmer says so or the actual function is known (i.e., not a virtual function), throws no exceptions, and only calls functions that are known to not throw any exceptions.
- Only local objects are automatically cleaned up.

Templates

Templates are pure compile-time features (i.e., no runtime overhead).

Caveat Emptors

- Template code that is not being used does not generate compiler errors and warnings like unused regular code does.
- Templates only form a meta-programming language by accident.
- A single template mistake can cascade into thousands of undecipherable error messages.