

Algorithms and techniques for a ready-to-run C++ software development environment

Craig D.M. Henderson
craig.henderson@virgin.net

March 2002

Traditional compiler techniques centre on individual source files in their entirety to form a *compilation unit*. Languages such as C and C++ define the scope of symbols relative to the compilation units and even provide specific keywords to manage the scope, such as `extern` and `static`. If a source file is modified in any way, then all the compilation units dependent on that file are recompiled to ensure the binary code is kept up to date. Much of the processing, analysis and code generation that a compiler performs for a single compilation unit is likely to be identical to the last time the unit was compiled. It follows, therefore, that if the dependencies could be defined with a finer granularity than that of an entire source file, then techniques could be developed to recompile partial source files relevant only to those changes that have been made. Indeed, some edits such as modifying white space or comment lines will require no compilation whatsoever, merely an update of line number information for debugging purposes.

Incremental Pre-processor

In his Ph.D. thesis “*Practical Algorithms for Incremental Software Development Environments*”, Wagner [1] describes a process of incremental lexical analysis utilising an existing batch lexical analyser, and incremental parsing using a traditional LALR(1) parser such as that generated by the well known and respected UNIX tool YACC. However, he does not investigate any improvements that could be made to the pre-processor. Is it possible to reduce repeated compilation based on symbol¹ changes? Can an incremental pre-processor eliminate the (relatively) recent concept of pre-compiled headers, or at least replace them as an efficient alternative?

- *Pre-processor elimination.* Can the traditional language pre-processor be integrated into the compilation process to eliminate extra processing and facilitate macro dependencies?
- *PP-Symbol level dependencies.* How can dependencies between individual symbols be managed?
- *Integration of file dependencies and with fine granularity dependencies.* Source file changes from outside of the development environment cannot be ignored. These changes can occur for any number of reasons such as updating files from a version control system or simply from using an external editor. At worst, these files must be identified as modified in their entirety, and processed in the traditional manner along with all dependent source files. However, a method of comparing files to identify new, deleted and modified symbols could be used to identify the changes and produce an input to the incremental compiler.

¹ The pre-processor works on a basis of lexical tokens. For the purposes of this text, the term *symbol* is used to define a lexical token that has been known to the pre-processor as having a special meaning or value.

Anticiparallelism

The CPU stays idle for most of the time and the developer must initiate the build process and then wait for it to complete. By taking a fundamentally different approach to compilation, couldn't a machine's resources be put to better use? In August 1998, Metcalfe [2] wrote of a concept he called *anticiparallelism*. "Today, 99 percent of computing cycles are wasted on idle time..." he explains, and continues to define anticiparallelism as a process that "looks for opportunities to deserialize computing" in order to "[keep] all available processors busy doing what users have requested" and "[anticipating] what users are likely to request next".

Research Areas

- *Load Balancing*. What algorithm will provide the most efficient use of the available CPU resource(s) to achieve background incremental compilation while remaining responsive to the user interface and the demands from other processes?
- *Change notification*. What methods can be used to notify changes to the source? At the file level (external file changes described above) and from an integrated environment.
- *IDE methodologies*. Communication between the editor user interface and the incremental compiler. When and how are changes to source propagated to the compiler? Algorithms for efficiency to minimise redundancy and optimise availability.

On-the-fly-updates

On-the-fly updates describe a mechanism for instance code updates. During a debugging session, code changes can be (incrementally) recompiled directly into the debuggee address space. Basic updates to code within an existing structural scope are relatively trivial and have been available in commercial compilers for many years, such as Microsoft Visual C++'s *Edit and Continue* feature. However, more significant changes to code are a greater challenge. This work is closely related to the increment compilation and fine granularity dependencies described above.

- *Pre-processor changes*. Changes to pre-processor statements represent particular difficulties with condition compilation and macro expansion.
- *Structural code changes* relating to the additional and removal of methods and object members, and the modification of function signatures and data types is especially difficult in an on-the-fly update situation. This will require a significant run-time overhead to manage the creation of heap objects so that they can be modified.

These on-the-fly updates are very difficult to achieve successfully in a manner that the C++ language standard is adhered to and is practicable in a professional development environment. This piece of research is the most likely to conclude that while the idea is valuable to the contribution of an improved development environment, the implementation is impractical or just simply wrong (within the bounds of the language).

Summary

By applying these theories to the compilation process, a development environment could be created to streamline the development process and minimise build times. The ultimate goal is to create a development environment that produces quality distributable binary code as the developer writes the source. This can be achieved with an incremental compiler running in the background, linked to the foreground editor user interface and will result in always-ready-to-run code and eliminates the hands-off build time that occupies so much of a developer's time.

References

- 1 Wagner, T.A., Practical Algorithms for Incremental Software Development Environments. March 1998
- 2 Bob Metcalf. From the Ether, InfoWorld magazine. August 10, 1998