Science & Technology Facilities Council

Technical Report

A Survey of C and C++ Software Tools for Computational Science

D.J. Worth, C. Greenough and L.S. Chin

December 2009

RAL-TR-2009-028

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

ISSN 1358-6254

RAL-TR-2009-028

A Survey of C and C++ Software Tools for Computational Science

D.J. Worth, C. Greenough and L.S. Chin

December 2009

Abstract

This report provides a survey of *some* of the software tools currently available to assist in the development of scientific software in C and C++. There are far more tools available for these languages than can be covered in this report and we come at the task of surveying the field with the eyes of *scientific software* developers. These developers do not work in the same way as commercial developers nor often to the same rules! The lack of a strict development model means that tools aimed at managing the complete lifecycle are missing from this report, however we will introduce the basic idea of the software lifecycle, its components and tools that could be used at each stage. There are sections covering coding (including source code quality assurance), source code control, the build environment, compiling, debugging, testing and source code documentation. We will also include material on libraries that we consider may be useful in the scientific software field.

Keywords: Scientific Software, Software Engineering, C, C++, Software Tools

---

Reports can be obtained from `www.softeng.cse.clrc.ac.uk`

Software Engineering Group
Computational Science & Engineering Department
STFC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
Oxfordshire OX11 0QX

Contents

1 Introduction

> *It is a truth universally acknowledged that a developer in possession of an algorithm must be in want of the Fortran programming language.*

Or, to put it another way, for many many years it has been held as self evident that efficient scientific software must be written in Fortran. While there are strong arguments in favour of this assumption: built in array datatypes, limitation to procedural coding style, better optimising compilers, availability of supporting numerical libraries among others, non of these is enough to reject other languages. The main reason Fortran continues to be used and the language evolve is the large amount of legacy code built up over the last decades. This code requires support and maintenance to keep it portable to new architectures and there are tools and techniques to help with this process described in [1] and the resources available at `http://www.softeng.cse.clrc.ac.uk/bpwiki/Home`.

We are not suggesting a wholesale switch of development language requiring re-engineering from Fortran to C++, that would be a waste of effort. When considering new software development however software groups are considering C and C++ as languages, e.g. the GNU Scientific Library (GSL) [2], SciMath [3], LAPACK++ [4] and CHARM++ [5]. These libraries are discussed further in section 6.

If scientific software developers are using or considering using C and C++ then it makes sense to use it properly, with good standards of software engineering and associated tools. After all what good are state-of-the-art algorithms and models and an all-consuming drive for the ultimate in efficiency when there is no evidence that the code does what it was required to, is easy for someone with the correct *scientific* background to maintain and develop, nor been tested in a thorough and planned way. Without this evidence agreement with experiment is nothing more than serendipity.

To address this point we give details about tools and techniques for good scientific software development with C and C++ staring in section 2. We will look at the areas of software design, implementation, testing and building software and offer a pragmatic approach in each that we hope will not leave the reader drowning in jargon, paperwork, and *the process*.

We complete the introduction by looking at the history of C and C++, their roles in scientific software and give a brief overview of the software development lifecycle. This latter section describes software engineering terms that are used throughout the report.

1.1 History of C and C++

There is a great deal of material available on-line regarding C and C++ so we only give a brief history and introduction to the two languages here.

C was originally developed at the Bell Telephone Laboratories in 1972 by Dennis Ritchie as a means of programming on the Unix operating system. It is a procedural language with the program split in to separate functions that are called to achieve the desired behaviour. It was originally designed and used for system software but has been widely used to develop applications software. Many compilers are available and it is supported on a wide array of architectures and platforms. It was not until 1989 that ANSI X3.159-1989 "Programming Language C" was ratified and this version of the language is often referred to as ANSI C, Standard C, or sometimes C89. The ISO adopted that standard a year later as ISO/IEC 9899:1990, sometimes called C90. A new standard (C99) was published as ISO/IEC 9899:1999 in 1999 and work is in progress on a third.

As an enhancement to C, C++ was developed by Bjarne Stroustrup in 1979 again at Bell Labs. The enhancements were in the area of *object-orientation* and added classes, inheritance,

virtual functions, operator overloading, templates and exception handling. The first standard (ISO/IEC 14882:1998) was adopted in 1998 and a second (ISO/IEC 14882:2003) in 2003. Work is in progress on a third. As an extension to C is is possible to program procedurally in C++.

1.2 C and C++ in Scientific Software

To date there has been little scientific application development in C or C++. There is a lot of software written in both languages and their popularity is demonstrated by the number of tools and libraries available in each language. We have already mentioned some scientific libraries developed in C and C++ and there are many more that we haven't. These are the building blocks for applications but it seems that those developing applications have largely stuck with what they (or their supervisors/project managers) know—Fortran. The issue of legacy software also supports the dominance of Fortran. New developments within existing software lead to new science and papers; re-engineering of existing code, whether by changing language or improving implementation in the existing language, does not.

The major argument against C is that is does not have native array data types. Arrays must be implemented as pointers with all the attendant memory management overhead(aches) they involve especially when multi-dimensional arrays are necessary. Until dynamic arrays appeared in Fortran 90, the memory management facilities of C were sometimes used to provide dynamic initial sizing of arrays in Fortran.

The object-oriented enhancements of C that produced C++ did nothing to improve the chances of the latter's adoption by the scientific community. All the overheads of virtual functions and operator overloading as well as the perceived insistence on *data hiding* made the knee jerk reaction one of "would not touch it with a long stick".

One of the author's experiences with C++ in the mid-90s has left him well aware of the problems of class proliferation, horrific levels of inheritance and abstraction for the sake of abstraction in which nothing must be seen to be doing any work.

There is little work on comparing performance of C or C++ against Fortran, the main reason being that no-one wants to re-engineer to a new language just to do a performance comparison. The nearest we can get is comparing algorithms in the area of numerical linear algebra (upon which much of scientific computing is based) and there is evidence that shows C++ can perform as well as Fortran with the right implementation [6]. The same author has put together a report on useful techniques for implementing scientific programs in C++, with an emphasis on using templates to improve performance [7].

1.3 Software Development

Many and varied are the descriptions of the software development process and its stages, just look up "Software development process" in Wikipedia. Ranging from the much derided *waterfall model* in which (formally) one stage leads to the next with no chance to revisit previous stages in the light of experience to the modern concepts of *extreme programming (XP)* and *test driven development* in which tasks are broken down into short iterations, there is little formal documentation and integration is continuous.

The common features of all the models are:

- Requirements
- Design
- Implementation

- Testing
- Release

In this document we will address in detail the middle four of these.

Requirements gathering is scientific software is usually a matter of one or two people deciding what would be interesting or necessary to include for their research. So long as these requirements are written down in a way that will make them useful when the process is complete it is not important how this is done. Requirements can be used as embryonic user and developer documentation and a basis for testing the whole system and should be understandable by code writers and those who will come after wanting to develop and maintain the code.

The release of software is also outside the scope of this document. All we will say is that software that is easy to install, or which may be complex but has good install instructions will stand more chance of being tried and used.

The details for design, implementation and testing are given in the following sections and following them we discuss compilers and build systems and some useful libraries that may reduce the work when developing scientific software.

2 Design

"Oh, no. I don't need design. I'll get on much quicker if I just start coding." This is one of the standard responses when the subject of software design is raised in scientific circles. Such an attitude is very short sighted. A single developer may get on more quickly by jumping straight to code but will be left floundering if they want to collaborate with other developers or explain how their software works. There is nothing wrong with wanting to get into coding early and the experienced software developer has a plan in his or her head. So why not take half a day making the design explicit? It is not against the rules to change things later, nor is change a sign of weakness.

There are many methods for writing down the design. A flow chart could be used for C or simply a textual list of the stages involved in the algorithm. For C++ there are a number of object oriented modelling languages that describe the classes and their interactions. Currently the most popular of these is the Unified Modeling Language (UML) [8] and its use can become extremely complicated—something that should be avoided in our pragmatic approach.

If we were asked to make recommendations then:

C Use a numbered list to write a kind of pseudo-code for the algorithm. Second level numbering could be used to expand some sections as required. Several lists could be used with one at the highest level and others for lower level detail.

C++ Use a class diagram to give an idea of the classes in the code and their relationships. Add the public methods (don't bother with constructors, set/get methods etc.) and important attributes for each class. Then write down how the software is built from these classes if it is not clear from the class diagram (which it might not be to other developers). The diagram should not be to complicated as it will be difficult to keep it up to date as the design changes during development.

The design can be written on paper or there are software tools that could be used. Some free tools are described in the rest of this section.

2.1 Dia

One tool that is fairly simple to use is Dia (`http://live.gnome.org/Dia`) available for Linux and Windows. It can produce flow charts and class diagrams in UML and has a simple code generation facility. Diagrams are loaded and saved to a custom XML format (gzipped by default, to save space) and Dia can export diagrams to a number of formats, including EPS, SVG, XFIG, WMF and PNG, and can print diagrams (including ones that span multiple pages).

There are a number of tools listed on the website that can take C++ code and create diagrams in Dia format.

2.2 Kivio

Kivio (`http://www.koffice.org/kivio/`) is part of KOffice for KDE and is available for Linux. It can produce flow charts and class diagrams in UML. A diagram can be saved in a number of formats and printed.

2.3 ArgoUML

ArgoUML (`http://argouml.tigris.org/`) includes support for all standard UML 1.4 diagrams. It runs on any Java platform and is available in ten languages. Diagrams can be saved as GIF, PNG, PostScript, Encapsulated PS, PGML and SVG. It also provides code generation for Java, C++, C#, PHP4 and PHP5. There are many other advanced features such as design critics and suggested corrections for problems identified in the design, "To Do" lists and checklists.

2.4 Xfig

An old favourite, Xfig (`http://www.xfig.org/`) contains drawing objects for flowcharts and UML class diagrams.

3 Implementation

This is where we really want to be. Writing code, running on big machines (that aren't really quite big enough), getting results and publishing papers. Just hold on, there is more to implementation than writing the code. It needs doing in a way that makes it easy to develop with collaborators, easy to maintain and port to new platforms and of good "quality".

3.1 Writing Code

This is an area that provokes flame wars, name calling and raised eyebrows and we are definitely not recommending one way of writing code over another, or one tool over another. There are broadly two types of tools here:

- Integrated development environments (IDEs) that include editor, GUI designer, source code control, compilation, debugging, running. The editor may include syntax highlighting, code auto-completion using type ahead suggestions, calltips for functions, bookmarks and refactoring support (changing variable names, extracting interface for abstract base class etc).
- Traditional editors that just edit text.

The division between these classifications is becoming increasingly blurred as the more traditional editors offer plugins to extend their functionality such as adding compilation from within the editor.

We will concentrate on IDEs here as they are problably less well known than the traditional editors such as `vi`, `(x)emacs`, `gedit`, `kate`, etc.

3.1.1 Anjuta DevStudio

This is a comprehensive GTK based IDE for various languages but has most functionality for C and C++ (`http://anjuta.sourceforge.net/features`). The features include

- File manager: Open file, compile, CVS/Subversion actions, project actions.
- Project manager: Open automake/autoconf project.
- Project wizard using autogen.
- Syntax highlighting, autoindentation, automatic code reformating (using indent).
- Code autocompletion: Automatic code completion for known symbols. Type ahead suggestions to choose for completion.
- Calltips for function prototypes: Provides helpful tips for typing function calls with their prototype and arguments hint.
- Code folding/hiding: Fold code blocks, functions, balanced texts to hide them in hierarchical order. Unfold to unhide them.
- Powerful search and replace: Supports string and regexp search expressions, search in files, search in project etc.
- Build messages highlight: Error/warning/information messages are indicated in the editor.
- Symbols view and navigation: Shows all symbols in your project organized into their types.
- Integrated debugger using gdb.
- Integrated glade (`http://glade.gnome.org/`) user interface designer.
- Valgrind plugin and gprof profiler plugin.

3.1.2 Code::Blocks

Code::Blocks (`http://www.codeblocks.org/`) is a free C++ IDE designed to be very extensible and fully configurable. It is built using wxWidgets and runs on Linux, Mac and Windows. Features include:

- Syntax highlighting
- Code folding
- Code completion
- Class browser
- Smart indent
- Multiple compiler support (GCC (MingW / GNU GCC), MSVC++, Digital Mars, Borland C++ 5.5, Open Watcom)
- Custom build system (no makefiles)
- Integrated debugger (GNU gdb and Microsoft CDB)

Other features are provided with plugins.

3.1.3 Eclipse CDT

The C/C++ development tool, CDT (`http://www.eclipse.org/cdt/`), is an Eclipse project providing a fully functional C and C++ Integrated Development Environment (IDE) for the Eclipse platform. It runs under Java on Linux, Mac and Windows and provides the following features:

- Syntax highlighting
- Project wizard and management
- Code assist: Pops up completion suggestions as you type
- Doxygen comment added for parameters and return types
- Code folding
- Code refactoring
- Index of entities in a file
- Compiling source code with error highlighting in the code
- Integrated debugger

The Eclipse platform has many other projects that enhance CDT such as integration of CVS/Subversion commands for files and GUI developers.

3.1.4 Geany

Geany (`http://www.geany.org/Main/About`) is a small and lightweight Integrated Development Environment. It was developed to provide a small and fast IDE, which has only a few dependencies from other packages. Another goal was to be as independent as possible from a special Desktop Environment like KDE or GNOME—Geany only requires the GTK2 runtime libraries and will run under Linux, Mac OS X and Windows.

Some basic features of Geany:

- Syntax highlighting
- Code folding
- Symbol name auto-completion
- Construct completion/snippets
- Call tips
- Symbol lists
- Code navigation
- Build system to compile and execute your code
- Simple project management

Other features are provided with plugins.

3.1.5 KDevelop

KDevelop (`http://www.kdevelop.org/`) is an IDE for KDE. Its feature set includes:

- Project management: Using automake, qmake, custom makefiles, ant for java projects
- Syntax highlighting
- On the fly syntax error indication

- Automatic code completion and code hinting for class variables, methods, function arguments and more.
- Class browser
- Code folding
- Code refactoring
- Integrated compilation
- Integrated debugger using gdb
- Integrated CVS and Subversion features
- GUI developer with QT
- Valgrind integration

3.1.6 Microsoft Visual Studio

The Microsoft IDE (`http://msdn.microsoft.com/en-gb/vs2008/products/cc149003.aspx`) runs on Windows and includes many features that are simply unnecessary for scientific software development. The free C++ Express edition (`http://www.microsoft.com/express/vc/`) may be worth a look but it has a strong slant towards the "visual" element and writing games.

3.1.7 Sun Studio

Sun Studio (`http://developers.sun.com/sunstudio/index.jsp`) is an IDE for C, C++ and Fortran that includes a great many tools for software development including parallelising compilers, code-level and memory debuggers, performance and thread analysis tools, OpenMP support as well as optimized maths libraries. The tools come packaged in a NetBeans based IDE running under Linux and Solaris with Java. There are two version one stable and the other (Sun Studio Express) showcasing the latest features.

Little specific to the source code editor can be gleaned from the website but from a quick trial we see it has:

- Syntax highlighting
- Project management
- Code completion
- CVS/Subversion integration
- Compilation with Sun's compiler
- Integrated debugging

3.1.8 XCode

XCode (`http://developer.apple.com/tools/xcode/index.html`) is Apple's IDE for Mac OS X. It is described as a graphical workbench that tightly integrates a professional text editor, a robust build system, a debugger, and the powerful GCC compiler capable of targeting Intel and PowerPC regardless of host platform.

The complete Mac OS X developer tools chain is distributed as part of Xcode; these tools include Interface Builder, Instruments, Dashcode, the WebObjects framework, and the complete reference documentation, to name just a few.

3.2 Source Code Control

As source code changes and versions of software distributed (either in source or compiled form) it is necessary to keep track of the changes made. Tracking changes made by collaborators is even more important, especially if those changes break existing functionality or mean code will not compile. One way of doing it is to keep separate directories for each version but no one has the discipline to do this correctly one their own, let alone when working in a team. The answer the source code control.

Source code control allows developers to track changes to the code over time, monitor who has changed what and when, undo changes and let a development team work in a controlled manner on the same source code (assuming no pathological behaviour!).

We will consider CVS and Subversion here because they are the two most widely used tools in this area and on this occasion recommend that Subversion is better. Distributed version control mechanisms will be mentioned briefly.

3.2.1 CVS

CVS (`http://www.nongnu.org/cvs/`) is a long standing source code control tool and is still widely used by software development teams with existing CVS repositories. CVS works in a client/server idiom with a central repository containing all the source code and version information on the server and developers (clients) checking out a particular version (usually the latest) from the repository to work on. When changes are complete they are committed to the repository with a log message. When two people have edited the same file a process of merging the second lot of changes may be necessary.

CVS stores the changes for each file in the repository but not the directories to which the files belong. This means files cannot be removed from a directory but empty files can be omitted when checking out code.

The concept of a module is very useful when a repository holds source code for individual components of a large piece of software in several directories. A module is a set of directories that can be checked out using the module name.

There are a number of CVS clients for various operating systems listed at
`http://ximbiot.com/cvs/wiki/CVS%20Clients`.

3.2.2 Subversion

Subversion (`http://subversion.tigris.org/`) has been described as a "better" CVS and has most of the features of CVS. It has extended CVS in the following ways:

- Directories and symbolic links have version information
- Copying, deleting and renaming are part of the version history
- Commits are atomic, i.e. a commit will fail if any part of it fails.
- Only changes are transferred during commits and updates, reducing network traffic, even for binary files.
- Standalone server option.

There are a number of Subversion clients for various operating systems listed at
`http://subversion.tigris.org/links.html#clients`.

3.2.3 Distributed Source Code Control

Both CVS and Subversion have a central repository and this is *the* repository, whereas distributed source code control has many repositories, possibly one for each developer. Repositories can be cloned and developers work separately without needing a network connection keeping their changes in their own repository, presumably committing frequently and then only commiting to somewhere more central when the code is ready. Independent work without the necessity of a network is more productive and this is one of the major advantages however a big disadvantage is that more complex merging will be required to produce a canonical version of the software.

This type of control seems to fit well with test driven development or extreme programming (XP) where commits are small and frequent but not necessarily of interest to the whole project until a feature is complete. It also chimes with the emphasis on communication between developers that is central to XP.

Some of the more popular tools in this area are:

- Bazaar — `http://www.bazaar-vcs.org/` — Linux, Mac, Windows
- Darcs — `http://darcs.net/` — Linux, Mac, Windows
- Git — `http://git.or.cz/` — Linux, Mac, Windows
- GNU Arch — `http://www.gnu.org/software/gnu-arch/` — Linux, Mac, Windows
- Mercurial — `http://www.selenic.com/mercurial/` — Linux, Mac, Windows
- Svk — `http://svk.bestpractical.com/` — Linux, Mac, Windows

3.3 Code Documentation

The concept of documenting code goes along with the concept of documenting the design and often gets the same response! Our response is the same. Think about those with whom you wish to collaborate and those who will maintain the code in the future. If software is to have a life then future developers will need to know what it does and how it does it. Comments in the code are the obvious solution but they must be relevant, useful and kept up to date. Equally important are comments on the purpose of a function or class and its attributes and methods. Again they need to be relevant, useful and kept up to date. In both cases "useful" means information that will help a new developer to gain a good understanding of how the code works so that they can get on with their work.

The tools listed in this section help by suggesting information that should go in such comments but cannot write the text. All the tools are designed to extract useful information (put in by the developer during development) and present it in an accessible form (e.g. HTML) for other developers or to help code re-use.

While not advocating coding before design we point out that there are tools available that can parse C++ code and create class diagrams to be read by Dia (see Section 2.1). They all require some tweaking of the resulting diagram in Dia but this is a good way of documenting the code. Tools are:

- AutoDia — `http://www.aarontrevena.co.uk/opensource/autodia/`
- Mendoosa — `http://medoosa.sourceforge.net/`
- cpp2dia — `http://cpp2dia.sourceforge.net/`

The search for a tool that can produce a call graph from C code has proved fruitless with those found either not compiling or not creating any output.

By far the most common way of documenting source code is by using special comment formats that can be parsed by a tool to create documentation to support and describe the code in an easy to navigate form such as HTML. The most well known and widely used of these is Doxygen and it is the recommended tool for documenting code.

3.3.1 Doxygen

We have used Doxygen (`http://www.stack.nl/~dimitri/doxygen/`) in our own projects and have found it very useful to provide supporting documentation, API documentation and developer documentation. It can handle C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, C#, and to some extent D and produce output in HTMLLaTeX, RTF, PostScript, hyperlinked PDF, compressed HTML, and Unix man pages.

It uses special comment markers and tags to provide information that appears in the documentation. General documentation as well as documentation for classes, methods, attributes, functions, data types among others is possible and it can use GraphViz to produce dependency graphs and inheritance diagrams if desired.

One useful thing to do is to configure Doxygen to extract the code structure from undocumented source files. This can make it easy to navigate a large body of source code for the first time.

3.3.2 Natural Docs

Rather than using special tags Natural Docs (`http://www.naturaldocs.org/`) attempts to have a more natural style of comments using keywords. It is a different approach and one we have not tried in our own projects. Basic support is available for C, C++ and Fortran among many others and it outputs HTML, either plain or using frames.

3.3.3 ROBODoc

ROBODoc (`http://www.xs4all.nl/~rfsber/Robo/robodoc.html`) can be used to document functions, methods, classes, variables, makefile entries, system tests, and many other things. It works with C, C++, Fortran, Perl and any other language that supports remarks. Documentation can be produced in HTML, XML DocBook, TROFF, ASCII, LaTeX or RTF format.

3.4 Static Analysis of Source Code

To some (including the authors in previous lives) compilation and execution are the ultimate test of good software. Agreement with experiment gives confidence that the software is in some sense correct but that is usually only for a favourite set of input files. Going beyond this is something that is becoming increasingly important as new platforms and novel architectures emerge. Any of these changes could trip up the software so it would be best to find out sooner than later—there will be less validation to be re-done.

One area to be addressed is therefore testing and there is a whole section on that coming up next. A second area in which tools can help is in analysis of source code itself. Since this does not require the code to be run it is often called *static analysis*. It goes beyond the efforts of a compiler and can check for

- Memory leaks.

- Unsafe practices — does the software present a security risk to the platform on which it is run.
- Coding conventions — style and layout.

Another element of static analysis is measuring the quality of software. Definitions of quality abound and we will not regurgitate them here except to say that quality is often a measure of how easy source code is to understand, maintain and develop. Quality is measured by metrics of which there are many hundred, some simple and others complex and there are special metrics for object oriented code. Our advise is that metrics should not be seen as the ultimate arbiter of good quality code. For example, if a function contains many simple `if-then-else` statements (e.g. printing an error message based on an error code) the complexity is high but it is easy to understand what is going on. Software developers must use their judgement to known when things need changing and when not.

The tools described in sections 3.4.1 to 3.4.3 concern checks on coding practices and those described in sections 3.4.4 to 3.4.6 concern software metrics. There are many commercial metrics tools available, among the most well known ones are:

- Understand for C/C++ by Scientific Toolworks Inc. `http://www.scitools.com/products/understand/`
- McCabe IQ by McCabe Software. `http://www.mccabe.com/iq.htm`
- Resource Standard Metrics by M Squared Technologies. `http://msquaredtechnologies.com/m2rsm/index.html`
- LDRA Testbed. `http://www.ldra.com/testbed.asp`.

3.4.1 Splint

Splint (`http://lclint.cs.virginia.edu/`) is a tool for statically checking C programs for security vulnerabilities and coding mistakes. With minimal effort, Splint can be used as a better lint. If additional effort is invested adding annotations to programs, Splint can perform stronger checking than can be done by any standard lint. It is a command line tool but a GUI has recently been released.

In our experience a straight forward execution of splint on source code has generated a very large amount of warnings and errors. This can be reduced by using the `-weak` flag which relaxes some conditions and only remarks on the most severe problems.

3.4.2 cppcheck

This program (`http://cppcheck.wiki.sourceforge.net/`) tries to detect bugs that your c/c++ compiler does not see. It is versatile and can check code that has non-standard code such as various compiler extensions, inline assembly code, etc. The standard checks are:

- Bad usage of memset/memcpy/memmove
- Memory leaks
- Buffer overruns
- Check class constructors (uninitialized member variables?)
- Using old functions that should be avoided such as ‘gets’ and ‘scanf’
- Invalid function usage. A few extra checks when using standard functions.

- Division with signed and unsigned operands
- Detect base classes that have non-virtual destructors

There are extra checks that can be enabled (but sometimes produce false positives) and other checks for things like unused code, usage of all struct members, operator= is correct.

3.4.3 Rough Auditing Tool for Security

RATS (`http://www.fortify.com/security-resources/rats.jsp`) is a tool for scanning C, C++, Perl, PHP and Python source code and flagging common security related programming errors such as buffer overflows and TOCTOU (Time Of Check, Time Of Use) race conditions.

RATS scanning tool provides a security analyst with a list of potential trouble spots on which to focus, along with describing the problem, and potentially suggest remedies. It also provides a relative assessment of the potential severity of each problem, to better help an auditor prioritize. This tool also performs some basic analysis to try to rule out conditions that are obviously not problems.

As its name implies, the tool performs only a rough analysis of source code. It will not find every error and will also find things that are not errors. Manual inspection of code is still necessary, but greatly aided with this tool.

3.4.4 C and C++ Code Counter

CCCC (`http://sourceforge.net/projects/cccc`) is a tool which analyses C++ and Java files and generates a report on various metrics of the code. Metrics supported include lines of code, McCabe's complexity [9] and metrics proposed by Chidamber and Kemerer [10] and Kafura and Henry [11].

Metrics are presented for the collection of files given, individual classes and their methods and C functions. The output is a collection of HTML pages that allow easy navigation of the results with colour coding of warning or danger levels for each metric and helpfully provide explanations of the metrics.

CCCC is our prefered tool for C and C++ metrics.

3.4.5 CppNcss

This tool (`http://cppncss.sourceforge.net/`) provides various measurements (also known as metrics) by statically analysing C++ source code, mainly aiming at evaluating maintainability. The features listed on its website include

- Non Commenting Source Statements (NCSS) and Cyclomatic Complexity Number (CCN) measurements [9]
- Measurements on file (declaration/implementation files) and function/method level
- Command line or Ant task driven
- Text or XML output
- XSL stylesheet for converting to HTML

3.4.6 Gnocchi

Gnocchi (`http://www.muftor.com/Wikka/wikka.php?wakka=Gnocchi`) is a complexity analyser for C++ code. It calculates complexity on a per function basis but instead of trying to parse C++ code Gnocchi reads the coverage information produced by GCC. If code is compiled with

`-fprofile-arcs` or `-ftest-coverage` (depending on compiler version) GCC creates a `.gcno` file for every object file. This is not so convenient as the other tools.

Gnocchi calculates cyclomatic [9] and the NPATH [12] complexity measures.

3.4.7 LDRA Testbed

This commercial tool is included here as it has components that cover static analysis, testing has a static analysis component (`http://www.ldra.com/staticanalysis.asp`). The analysis results include:

- Programming Standards Verification. Assesses whether the source code conforms to a set of user-configurable programming standards.
- Structured Programming Verification. Reports on whether the source code is properly structured.
- Complexity Metric Production. Reports on a number of complexity metrics such as Cyclomatic Complexity, Knots, Essential Cyclomatic Complexity, Essential Knots and many more.
- Full Variable Cross Reference. Examines and reports global and local variable usage within and across procedures and file boundaries.
- Unreachable Code Reporting. Reports on areas of redundant code.
- Static Data Flow Analysis. Follows variables through the source code and reports any anomalous use.
- Information Flow Analysis. Analyses inter-dependencies of variables for all paths through the code.
- Loop Analysis. Reports the looping structure and depth of nesting within the code.
- Analysis of Recursive Procedures. All the analysis above is performed individually and on sets of mutually recursive procedures.
- Procedure Interface Analysis. The interface for each procedure is analysed for defects and deficiencies. The interfaces are then projected through the call graph of a system to highlight integration defects.

4 Testing

Now the source code is written, with useful comments, with reasonable metric values and it is safely stored in a source code repository. It compiles using gcc and there are no apparent memory leaks or other problems because splint says so. It runs on the favourite set of test problems and gives the right answer.

> The next step is testing ... *But hang on, it was tested with the test problems.*
>
> Are these tests sufficient?
>
> *What do you mean?*
>
> Well, does it handle invalid input in a helpful way? does it fail gracefully? does each component work correctly? what about each function or method?
>
> *I see, please tell me more.*

We are looking at two areas here:

- Testing components of the program in isolation to ensure that they work correctly. If external libraries are used then we assume that they are correct or run their supplied tests. Trivial functions or class methods can be visually inspected for correctness but more complex functions or methods should be tested to ensure that they work correctly. This is known as *unit testing* and we describe tools in Section 4.1.
- Testing the the program as a whole. This covers things such as validation against experimental results, verification that it works correctly with invalid input, useful error handling, ensuring that important sections of the code are actually tested, and performance testing. This is known as *integration* or *system testing* and tools are described in Section 4.2

The Software Engineering Group has produced a detailed report testing including sections on test management and testing tools [13].

4.1 Unit testing

As we mentioned before, unit testing is testing individual components of the source code to make sure they work as we intend. This is useful in a number of ways: to provide confidence that a numerical algorithm has been coded correctly, to demonstrate that a physical properties component produces the correct results, to test a new algorithm or new implementation of the component quickly against previous results, it is easier to do coverage testing for the component in isolation rather than as part of the whole.

There is a great deal of material on unit testing available on-line and we will not go into detail here except to explain that there are three section to a unit test:

1. Set up supporting data structures, classes and input data for the test (known as the *fixture*).
2. Run the function/method and test the results.
3. Free memory and delete data no longer required (known as *tear down*).

Individual unit tests are built into test suites and summary results of the suites reported at the end of the testing process. There are tools to help developers perform unit testing.

4.1.1 CUnit

CUnit (`http://cunit.sourceforge.net/`) is a lightweight system for writing, administering, and running unit tests in C. It provides C programmers a basic testing functionality with a flexible variety of user interfaces. It is built as a static library which is linked with the user's testing code. It uses a simple framework for building test structures, and provides a rich set of assertions for testing common data types. In addition, several different interfaces are provided for running tests and reporting results. These interfaces currently include: automated output to XML, basic non-interactive interface, interactive console interface, and interactive Curses based interface.

We have used this tool in non-interactive mode and found that it is straight forward to set up suites of unit tests and run them as part of on-going software development and as a regression testing tool.

4.1.2 CppUnit

CppUnit (`http://apps.sourceforge.net/mediawiki/cppunit/index.php?title=Main_Page`) is a C++ unit testing framework. It started its life as a port of JUnit to C++ and has been developed with the following features:

- XML output with hooks for additional data (XSL format available in release 1.10.2 needs some fixing)
- Compiler-like text output to integrate with an IDE
- Helper macros for easier test suite declaration
- Hierarchical test fixture support
- Test registry to reduce recompilation need
- Test plug-in for faster compile/test cycle (self testable dynamic library)
- Protector to encapsulate test execution (allow capture of exception not derived from std::exception)

Code is included to build applications to run the tests with a variety of graphical toolkits: MFC, QT, Curses and WxWidgets.

4.2 Integration and System Testing

In this section we will discuss how to test the whole executable version of the code to ensure that it is in the best state to be distributed to users. The users will want to know that it has been tested and show to get the correct answers and also that it will run their data sets.

The first of these is sometimes referred to as *validation testing* and there should be a number of test cases that produce results that can be checked against experiment or known solutions. To help in this area there are tools that can compare files of numbers in the same way as the unix `diff` command but with a tolerance within which two real numbers are considered equal. These tools are also necessary when comparing the results of one version of the software with a newer version to ensure that no bugs have been introduced[1], known as *regression testing*.

Ensuring that the software will run on a user's data sets is a little more complex. It is often called *verification testing* and means testing that the software does what the input tells it to do and that it can detect invalid input. Knowing what code the software executes for given inputs falls under the label of *coverage analysis* and there are a number of tools that can show the code statements executed during a run of the software. A debugger could be used for this but leaves no record of the lines executed. Extending coverage testing over all data sets will build up a picture of which portions of the code have not been executed and could contain unforeseen problems once users get their hands on the code. Adding a new data set to run one of these portions will increase confidence.

A second use for coverage analysis is to see lines or portions of code that are executed a very large number of times. These are potential areas for developer inspired rather that compiler produced optimisation. *Performance analysis* tools are available to provide timings either at a function or line level to help see exactly how much time is spent in the various parts of the code.

While not strictly testing tools, debuggers tend to be used during the testing phase of development when the numerical results are not those expected, the software does not behave as expected, or crashes inexplicably. In C and C++ programs, where memory management is the developer's responsibility, errors can be down to use of uninitialised memory, pointer errors, doubly freeing a pointer, and memory leaks. Tools to help identify such problems are also described here.

The rest of this section discusses the tools for these forms of testing as follows:

- Validation with numerical differencing — Sections 4.2.1 to 4.2.3.
- Coverage analysis — Sections 4.2.4 to 4.2.6.

[1] Of course, if the original results were erroneous because of a bug that has been fixed a different test is required.

- Performance analysis — Sections 4.2.7 to 4.2.11.
- Debuggers, memory analysis etc — Sections 4.2.12 to 4.2.17.

We mention the commercial tool LDRA Testbed (`http://www.ldra.com/testbed.asp`) here as it has a coverage analysis component in addition to a static analysis component which may make it a worthwhile investment for serious C/C++ developers.

4.2.1 Ndiff

Ndiff (`http://www.math.utah.edu/~beebe/software/ndiff/`) assumes that you have two text files containing numerical values, and the two files are expected to be identical, or at least numerically similar. ndiff allows you to specify absolute and/or relative error tolerances for differences between numerical values in the two files, and then reports only the lines with values exceeding those tolerances. It also tells you by how much they differ.

4.2.2 Numdiff

Numdiff (`http://www.nongnu.org/numdiff/`) is a program that can be used to compare putatively similar files line by line and field by field, ignoring small numeric differences and/or different numeric formats. It can also carry out the usual text differencing operations. By default, Numdiff assumes the fields are separated by white spaces (blanks, horizontal tabulations and newlines), but the user can also specify its list of separators through a command line option.

4.2.3 Toldiff

Tolerant file differencing tool Toldiff (`http://sourceforge.net/projects/toldiff/`) is a diff tool that allows tolerable (insignificant) differences between two files to be suppressed showing only the important ones. The tolerable differences are recorded running the tool with an appropriate command line flag.

4.2.4 GNU gcov

The gcov tool (`http://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html`) is part of the GNU gcc compiler suite and works with code compiled with gcc using special flags. It allows a developer to discover how often each line of code is executed and which lines of code actually are executed.

4.2.5 ggcov

Ggcov (`http://ggcov.sourceforge.net/`) is a GTK+ GUI for exploring test coverage data produced by C and C++ programs compiled with gcc. It is a GUI replacement for gcov but note that ggcov is not a front end for gcov; instead it reads the same data files directly and does various extra processing on them.

ggcov also has a web interface which provides a large fraction of the functionality of the GTK+ GUI.

4.2.6 lcov

LCOV (`http://ltp.sourceforge.net/coverage/lcov.php`) is an extension of GCOV consisting of a set of PERL scripts which build on the textual GCOV output to implement the following enhanced functionality:

- HTML based output: coverage rates are additionally indicated using bar graphs and specific colours.
- Support for large projects: overview pages allow quick browsing of coverage data by providing three levels of detail: directory view, file view and source code view.

LCOV was initially designed to support Linux kernel coverage measurements, but works as well for coverage measurements on standard user space applications.

4.2.7 gprof

The GNU tool gprof (`http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`) requires that the executable has been compiled with the GNU C/C++ compiler using certain options. Running the executable will then generate profiling data which is analysed using gprof.

The result of the analysis is a file containing two tables, the flat profile and the call graph (plus text which briefly explain the contents of these tables).

The flat profile shows how much time the program spent in each function, and how many times that function was called. If the information required is simply which functions burn most of the cycles, it is stated concisely here.

The call graph shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. This can suggest places where it would be worth trying to eliminate function calls that use a lot of time.

4.2.8 oprofile

OProfile (`http://oprofile.sourceforge.net/news/`) is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. OProfile is released under the GNU GPL.

It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information.

OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications. Profile data can be produced on the function-level or instruction-level detail. Source trees annotated with profile information can be created. A hit list of applications and functions that take the most time across the whole system can be produced.

4.2.9 Google Perf Tools

(`http://code.google.com/p/google-perftools/`) These tools are for use by developers so that they can create more robust applications. Especially of use to those developing multi-threaded applications in C++ with templates. Includes TCMalloc, heap-checker, heap-profiler and cpu-profiler. Perf Tools is distributed under the terms of the BSD License.

4.2.10 Sun Studio

The Sun Studio performance analysis tool (`http://developers.sun.com/sunstudio/index.jsp`) comprises a Collector and a Performance Analyzer used to collect and analyse performance data for an application. Both tools can be used from the command line or from a graphical user interface.

The Collector and Performance Analyzer are designed for use by any software developer, even if performance tuning is not the developers main responsibility. These tools provide a more flexible, detailed, and accurate analysis than the commonly used profiling tools prof and gprof, and are not subject to an attribution error in gprof.

The Collector and Performance Analyzer tools help to answer the following kinds of questions:

- How much of the available resources does the program consume?
- Which functions or load objects are consuming the most resources?
- Which source lines and instructions are responsible for resource consumption?
- How did the program arrive at this point in the execution?
- Which resources are being consumed by a function or load object?

4.2.11 Intel VTune

(`http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm`) The VTune Performance Analyzer from Intel makes application performance tuning easier with a graphical user interface and no recompiles required. It is compiler and language independent so it works with C, C++, Fortran, C#, Java, .NET and more. Unlike some products that offer only call graph analysis or only a limited set of sampling events, VTune analyzer offers both with an extensive set of tuning events for all the latest Intel processors. It is available for both Linux and Windows platforms.

4.2.12 The GNU Project Debugger

GDB (`http://www.gnu.org/software/gdb/gdb.html`) comes with the GNU C/C++ compiler. It allows you to see what is going on 'inside' another program while it executes—or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start a program, specifying anything that might affect its behaviour.
- Make a program stop on specified conditions.
- Examine what has happened when the program has stopped.
- Change things in the program, thus correcting the effects of one bug (which will need correcting in the source code) and go on to learn about another.

The program being debugged can be written in Ada, C, C++, Objective-C, Pascal (and many other languages). Those programs might be executing on the same machine as GDB (native) or on another machine (remote). GDB can run on most popular UNIX and Microsoft Windows variants.

4.2.13 DDD

GNU DDD (`http://www.gnu.org/software/ddd/`) is a graphical front-end for command-line debuggers such as GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, the bash debugger bashdb, the GNU Make debugger remake, or the Python debugger pydb. Besides "usual" front-end features such as viewing source texts, DDD has become famous through its interactive graphical data display, where data structures are displayed as graphs.

4.2.14 Nemiver

Nemiver (`http://projects.gnome.org/nemiver/`) is an on-going effort to write a standalone graphical debugger that integrates well in the GNOME desktop environment. It currently features a back end which uses the well known GNU Debugger gdb to debug C / C++ programs. It allows the user to set breakpoints, view values of variables and step through the code.

4.2.15 Integrated Debuggers

The following IDEs have integrated debuggers.

- Sun Studio — dbx
- Microsoft Visual Studio
- Eclipse CDT — gdb back-end
- Anjuta — gdb back-end
- Code::Blocks — gdb and Microsoft cdb back-ends
- KDevelop — gdb back-end

4.2.16 Mudflap

Mudflap is a pointer debugging system included in the gcc compiler. It is enabled by passing `-fmudflap` to the compiler. For front-ends that support it (C and very simple C++ programs), it instruments all risky pointer/array dereferencing operations, some standard library string/heap functions, and some other associated constructs with range/validity tests. Modules so instrumented should be immune to buffer overflows, invalid heap use, and some other classes of C/C++ programming errors. The instrumentation relies on a separate runtime library (libmudflap), which will be linked into a program if `-fmudflap -lmudflap` is given at link time. Run-time behaviour of the instrumented program is controlled by the `MUDFLAP_OPTIONS` environment variable.

Details of the options can be found at `http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging`.

4.2.17 Valgrind

Valgrind (`http://valgrind.org/`) is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile programs in detail. It can also used to build new tools.

The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache profiler, and a heap profiler. It also includes one experimental tool, which detects out of bounds reads and writes of stack, global and heap arrays. It runs on the following platforms: X86/Linux, AMD64/Linux, PPC32/Linux, PPC64/Linux.

5 Compilation and Build Systems

Compilation is the act of turning the source code into a machine executable application. A compiler is used for this task and, unless there is only one source file (*usually bad, very very bad!*), the compiler will need to be run on all the source files and then on the resulting object files to create the executable. No-one would do this manually and so there are a number of build

systems that can ease this task by knowing how to compile to object file and executable and knowing which object files need updating and which not.

5.1 Compilers

Compiling source code to an application is something so routine that it becomes forgotten in the development cycle. For many there is no need to look beyond the GNU compilers that come with Linux or MinGW or Cygwin but some of the commercial products may offer something that could be useful. We will not present a comparison between compilers here just the bare details for a number of compilers.

5.1.1 GNU Compiler

Gcc is the standard compiler available in Linux and is a C compiler. The companion C++ compiler is g++. Details on the compiler can be found at `http://gcc.gnu.org/onlinedocs/` and the gcc website is `http://gcc.gnu.org/`.

Versions of gcc and g++ can be installed on Windows by making use of cygwin [14] and MinGW [15].

5.1.2 Intel Compiler

`http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm`. This website is the starting point for information on Intel compilers. All we will say here is that version are available for Windows to run under MS Visual Studio, Linux to run under Eclipse and Mac OS X to run under XCode.

5.1.3 Borland Compiler

Borland offer a full Windows IDE as a commercial product but the command line version of their Windows C++ compiler (version 5.5) and associated tools can be downloaded free. Use the following website for details `http://www.codegear.com/downloads/free/cppbuilder`.

5.1.4 Digital Mars Compiler

Digital Mars (`http://www.digitalmars.com/`) offer a C/C++ compiler. They claim the fastest compile/link times in the industry and powerful optimization technology with advanced register allocation and instruction scheduling.

5.1.5 IBM Compiler

IBM (`http://www-01.ibm.com/software/awdtools/xlcpp/`) offer C and C++ compilers for Linux (IBM Power Systems, including the POWER6 processors), AIX, Blue Gene, z/OS, OS/390 and z/VM.

5.1.6 Microsoft Visual C++

The compiler comes as part of the IDE —
`http://msdn.microsoft.com/en-gb/vs2008/products/cc149003.aspx`.

5.1.7 Sun Compiler

The compiler comes as part of Sun Studio (`http://developers.sun.com/sunstudio/index.jsp`) and has the following highlighted features:

- Auto-parallelisation of single-threaded code
- Static data-race and deadlock-detection for x86
- Option for Thread Analyzer support
- Compiler flags to optimize for multi-core architectures

5.1.8 XCode

XCode, Apple's IDE for Mac OS X (`http://developer.apple.com/tools/xcode/index.html`) comes with a version of the gcc compiler but can be used with the Intel compiler.

5.2 Build systems

These systems are designed to let the developer encode the steps to compile the application so that these steps do not have to be remembered. In addition they can shorten the compile time by working out which source files have changed and only compiling the files affected by these changes. Some IDEs have their own build systems and hide the details from the user while still allowing access to flags or options for debug symbols, language compliance, optimisation etc.

Most of the systems use the idea of *targets* to simplify the command for building the application. The developer does not have to remember the application's name, he or she can say "build everything", or "build the application so I can debug it" or "run the analysis tools" or "build the application and run the tests". This is a good way of automating analysis and testing.

Let us star with the autotools designed to ease the process of compiling and installing software then move on to the various build systems.

5.2.1 The Autotools

The autotools are a set of tools designed to make compilation and installation of software easy on multiple platforms using a simple set of commands. They search the target platform for the libraries and headers required by the software and take instruction on where the software is to be installed. Finally they generate makefiles (see Section 5.2.2) to build and install the software. If a developer makes use of these tools then the user should only need to type three commands:

```
% ./configure
% make
% make install
```

The autotools are:

autoconf `http://www.gnu.org/software/autoconf/` Autoconf creates a configuration script for a package from a template file that lists the operating system features that the package can use, in the form of M4 macro calls.

automake `http://www.gnu.org/software/automake/` Automake is a tool for automatically generating 'Makefile.in' files compliant with the GNU Coding Standards. Automake requires the use of Autoconf.

libtool `http://www.gnu.org/software/libtool/` GNU libtool is a generic library support script. Libtool hides the complexity of using shared libraries behind a consistent, portable interface.

A good tutorial on these useful but rather complex tools can be found at
`http://www.freesoftwaremagazine.com/books/autotools_a_guide_to_autoconf_automake_libtool`.

5.2.2 Make

This is the original build system used routinely to build major software projects over many years and still the most popular. It is available on Linux, Windows (via cygwin [14] or MinGW [15]) and Mac OS X. There is no definitive make website but a good introduction can be found at the GNU Make site `http://www.gnu.org/software/make/`.

Make gets its knowledge of how to build your program from a file called the *makefile*, which lists each of the non-source files (including the executables) and how to compute each of them from other (source) files. Developers should consider the makefile part of their source code.

Some of the capabilities of make are:

- Make enables an end user to build and install the application without knowing the details of how that is done because these details are recorded in the makefile.
- Make figures out automatically which files it needs to update, based on which source files have changed. It also automatically determines the proper order for updating files, in case one non-source file depends on another non-source file.
- Make is not limited to any particular language. For each non-source file in the program, the makefile specifies the shell commands to compute it. These shell commands can run a compiler to produce an object file, the linker to produce an executable, ar to update a library, or TeX or Makeinfo to format documentation.
- Make is not limited to building a package. It can also be used to control installing or uninstalling a package, generate tag tables for it, or anything else done often enough to make it worth while writing down how to do it.

5.2.3 CMake

Cross platform make (`http://www.cmake.org/`) is a family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of choice.

CMake is designed to support complex directory hierarchies and applications dependent on several libraries. For example, CMake supports projects consisting of multiple toolkits (i.e., libraries), where each toolkit might contain several directories, and the application depends on the toolkits plus additional code. CMake can also handle situations where executables must be built in order to generate code that is then compiled and linked into a final application.

5.2.4 SCons

SCons (`http://www.scons.org/`) is an Open Source software construction tool—that is, a next-generation build tool. Think of SCons as an improved, cross-platform substitute for the classic Make utility with integrated functionality similar to autoconf/automake and compiler caches such as ccache. In short, SCons is an easier, more reliable and faster way to build software.

It is known to work on Linux, other POSIX systems (including AIX, *BSD systems, HP/UX, IRIX and Solaris), Windows NT, Mac OS X, and OS/2 and features include:

- Configuration files are Python scripts—use the power of a real programming language to solve build problems.
- Reliable, automatic dependency analysis built-in for C, C++ and Fortran—no more "make depend" or "make clean" to get all of the dependencies. Dependency analysis is easily extensible through user-defined dependency Scanners for other languages or file types.
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt and SWIG, and building TeX and LaTeX documents. Easily extensible through user-defined Builders for other languages or file types.
- Integrated Autoconf-like support for finding #include files, libraries, functions and typedefs.

5.2.5 Jam

Jam (`http://www.perforce.com/jam/jam.html`) (to go with SCons presumably!) is a software build tool that makes building simple things simple and building complicated things manageable. It has been freely available as C source for many years and is widely used to build commercial and academic software. Jam is a very good solution for conventional C/C++ compile-and-link builds.

Because Jam understands C/C++ dependencies, there is no need to declare header or object files. The built-in Jam rule "Main" handles header file dependencies and object files both automatically and on-the-fly.

Some of the features are:

- Jam is small, it has negligible CPU overhead, and it does not create or leave behind temporary files.
- Jam is able to build large projects spread across many directories in a single pass and can manage and distribute build steps to multiple processors on one or more networked machines.
- Jam runs on UNIX, VMS, NT, OS/2, Macintosh OS X, and Macintosh MPW. Most Jamfiles are also portable.
- Platform independent rules and platform specific actions can be defined separately from dependency rules.
- Developers can enhance/extend Jam by creating user defined rules to utilize other built-in directives.
- Jam includes flow-control statements, variables, and a few other features of general purpose languages.

6 Useful Libraries

There are so many C and C++ libraries designed to provide useful functionality for the software developer that it would be impossible to cover even all the technological areas let alone the individual libraries. In this section we will concentrate on our selection for scientific software developers, limited (?) to the following areas:

- Numerical algorithms
- Visualisation

- Other general libraries

A Larger list can be found at on Mathtools.net [16]

6.1 Numerical Libraries

At the heart of the vast majority of scientific computing are numerical algorithms, for linear algebra, solving differential equations or integral equations, and performing Fourier transforms. The following selection of libraries range from the very simple class libraries for matrix objects and operations to the complete libraries of numerical algorithms. A larger list can be found on the Object-Oriented Numerics Page [17].

6.1.1 The Matrix Template Library

The Matrix Template Library (MTL—`http://www.osl.iu.edu/research/mtl/`) is a high-performance generic component library that provides comprehensive linear algebra functionality for a wide variety of matrix formats.

As with the Standard Template Library (STL), MTL uses a five-fold approach, consisting of generic functions, containers, iterators, adaptors, and function objects, all developed specifically for high performance numerical linear algebra. Within this framework, MTL provides generic algorithms corresponding to the mathematical operations that define linear algebra. Similarly, the containers, adaptors, and iterators are used to represent and to manipulate concrete linear algebra objects such as matrices and vectors. MTL includes a large number of data formats and algorithms, including most popular sparse and dense matrix formats and functionality equivalent to Level 3 BLAS.

To many scientific computing users, however, the advantages of an elegant programming interface are secondary to issues of performance. Generic programming is a powerful tool in this regard as well - performance tuning can itself be described in a generic fashion. These performance tuning abstractions are realized in a generic low-level library - the Basic Linear Algebra Instruction Set (BLAIS). Experimental results show that MTL with the BLAIS achieves performance that is as good as, or better than, vendor-tuned libraries. Thus, MTL demonstrates that the proper abstractions can be used to achieve high levels of performance, contrary to conventional wisdom. In addition, MTL requires orders of magnitude fewer lines of code for its implementation, with the concomitant savings in development and maintenance effort.

6.1.2 LAPACK++

LAPACK++ (`http://sourceforge.net/projects/lapackpp/`) is a library for high performance linear algebra computations. This version includes support for solving linear systems using LU, Cholesky, QR matrix factorizations, for real and complex matrices.

6.1.3 SparseLib++

SparseLib++ (`http://math.nist.gov/sparselib++/`) is a C++ class library for efficient sparse matrix computations across various computational platforms. The software package consists of matrix classes encompassing several sparse storage formats (e.g. compressed row, compressed column and coordinate formats), and providing basic functionality for managing sparse matrices. The Sparse BLAS Toolkit is used to for efficient kernel mathematical operations (e.g. sparse matrix-vector multiply) and to enhance portability and performance across a wide range of computer architectures. Included in the package are various preconditioners commonly used

in iterative solvers for linear systems of equations. The focus is on computational support for iterative methods (for example, see IML++), but the sparse matrix objects presented here can be used in their own right.

SparseLib++ matrices can be built out of nearly any C++ matrix/vector classes; it is shipped with the MV++ classes by default.

6.1.4 Iterative Methods Library in C++

This library (`http://math.nist.gov/iml++/`) is a C++ template based library of modern iterative methods for solving both symmetric and nonsymmetric linear systems of equations. It is a companion library to SparseLib++ and MV++. The algorithms are fully templated in that the same source code works for dense, sparse, and distributed matrices. Methods implemented include:

- Richardson Iteration
- Chebyshev Iteration
- Conjugate Gradient (CG)
- Conjugate Gradient Squared (CGS)
- BiConjugate Gradient (BiCG)
- BiConjugate Gradient Stabilized (BiCGSTAB)
- Generalized Minimum Residual (GMRES)
- Quasi-Minimal Residual Without Lookahead (QMR)

6.1.5 Numerical Matrix/Vector Classes in C++

MV++ (`http://math.nist.gov/mv++/`) is a small, efficient, set of concrete vector and simple matrix classes for numerical computing. It is not intended as a general vector container class, but rather designed specifically for optimized numerical computations on RISC and pipelined architectures. It is one step above a C/C++ array. The various MV++ classes form the building blocks of larger user-level libraries such as SparseLib++.

6.1.6 Template Numerical Toolkit

TNT (`http://math.nist.gov/tnt/index.html`) as it is known, is a collection of interfaces and reference implementations of numerical objects useful for scientific computing in C++. The toolkit defines interfaces for basic data structures, such as multidimensional arrays and sparse matrices, commonly used in numerical applications. The goal of this package is to provide reusable software components that address many of the portability and maintenance problems with C++ codes.

TNT provides a distinction between interfaces and implementations of TNT components. For example, there is a TNT interface for two-dimensional arrays which describes how individual elements are accessed and how certain information, such as the array dimensions, can be used in algorithms; however, there can be several implementations of such an interface: one that uses expression templates, or one that uses BLAS kernels, or another that is instrumented to provide debugging information. By specifying only the interface, applications codes may utilize such algorithms, while giving library developers the greatest flexibility in employing optimization or portability strategies.

6.1.7 ARPACK++

ARPACK++ (`http://www.caam.rice.edu/software/ARPACK/arpack++.html`) is an object-oriented version of the ARPACK package. It consists a collection of classes that offers C++ programmers an interface to ARPACK. It preserves the full capability, performance, accuracy and low memory requirements of the FORTRAN package, but takes advantage of the C++ object-oriented programming environment.

Some of the features included in ARPACK++ are:

- The use of templates to reduce the work needed to establish and solve eigenvalue problems and to simplify the structure utilized to handle such problems.
- The ability to easily find interior eigenvalues and to solve generalized problems. ARPACK++ includes several matrix classes that use routines from SuperLU, UMFPACK and LAPACK to solve linear systems. With these classes, spectral transformations such as the shift and invert method can be employed to find internal eigenvalues of regular and generalized problems without requiring the user to explicitly solve linear systems.
- A friendly interface that avoids the complication of the reverse communication interface that characterizes the FORTRAN version of ARPACK. For instance, the user can supply the non-zero elements of a matrix and obtain its eigenvalues and eigenvectors stored using the Standard Template Library (STL) vector class.
- A structure that minimizes the work needed to generate an interface between ARPACK and other libraries, such as the Template Numerical Toolkit (TNT) (see 6.1.6.

6.1.8 Boost Basic Linear Algebra

uBLAS (`http://www.boost.org/doc/libs/1_37_0/libs/numeric/ublas/doc/index.htm`) is the Boost basic linear algebra library. It is a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices. The design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates. Functionality

uBLAS provides templated C++ classes for dense, unit and sparse vectors, dense, identity, triangular, banded, symmetric, hermitian and sparse matrices. Views into vectors and matrices can be constructed via ranges or slices and adaptor classes. The library covers the usual basic linear algebra operations on vectors and matrices: reductions like different norms, addition and subtraction of vectors and matrices and multiplication with a scalar, inner and outer products of vectors, matrix vector and matrix matrix products and triangular solver. The glue between containers, views and expression templated operations is a mostly STL conforming iterator interface.

6.1.9 GNU Scientific Library

The GNU Scientific Library (GSL—`http://www.gnu.org/software/gsl/`) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.

The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite. Take a look at the website for the details.

6.1.10 Blitz++

Blitz++ (`http://www.oonumerics.org/blitz/`) is a C++ class library for scientific computing which provides performance on par with Fortran 77/90. It uses template techniques to achieve

high performance. The current versions provide dense arrays and vectors, random number generators, and small vectors and matrices. Blitz++ is distributed freely under an open source license, and contributions to the library are welcomed.

6.1.11 PETSc

PETSc (`http://www.mcs.anl.gov/petsc/petsc-as/`), is a suite of data structures and routines written in C for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication. Features include:

- Parallel vectors: includes code for communicating ghost points
- Parallel matrices: several sparse storage formats, easy, efficient assembly.
- Scalable parallel preconditioners
- Krylov subspace methods
- Parallel Newton-based nonlinear solvers
- Parallel timestepping (ODE) solvers

6.1.12 Getfem++

The Getfem++ project (`http://home.gna.org/getfem/`) focuses on the development of a generic and efficient C++ library for finite element methods. The goal is to provide a library allowing the computation of any elementary matrix (even for mixed finite element methods) on the largest class of methods and elements, and for arbitrary dimension (i.e. not only 2D and 3D problems).

It offers a complete separation between integration methods (exact or approximated), geometric transformations (linear or not) and finite element methods of arbitrary degrees. It can really relieve a more integrated finite element code of technical difficulties of elementary computations.

The library also includes the usual tools for finite elements such as assembly procedures for classical PDEs, interpolation methods, computation of norms, mesh operations (including automatic refinement), boundary conditions, post-processing tools such as extraction of slices from a mesh ...

Getfem++ has no meshing capabilities (apart regular meshes and a small attempt), hence it is necessary to import meshes. Imports formats currently known by getfem are GiD , GmSH and emc2 mesh files. However, given a mesh, it is possible to refine it automatically.

6.1.13 Overture

Overture (`https://computation.llnl.gov/casc/Overture/`) is an object-oriented code framework for solving partial differential equations. It provides a portable, flexible software development environment for applications that involve the simulation of physical processes in complex moving geometry . It is implemented as a collection of C++ libraries that enable the use of finite difference and finite volume methods at a level that hides the details of the associated data structures. Overture is designed for solving problems on a structured grid or a collection of structured grids. In particular, it can use curvilinear grids, adaptive mesh refinement, and the composite overlapping grid method to represent problems involving complex domains with moving components. Techniques for building grids on CAD geometries and for building hybrid grids that can be used with applications that use unstructured grids have also been developed.

6.1.14 FFTPACK++

(`http://programming.ccp14.ac.uk/mumit-khan/~khan/software/fftpack/fftpack.html`) FFTPack++ is a C++ wrapper for FFTPACK complex routines using LAPACK++ Matrix and Vector classes. FFTPACK routines were converted to C using f2c and also modified to use double precision complex using -r8 to f2c.

6.1.15 The NAG C Library

The NAG C Library (`http://www.nag.co.uk/numeric/CL/CLdescription.asp`) is a commercial library described as "the largest and most comprehensive collection of mathematical and statistical algorithms for C and C++ programmers available today". Organizations all over the world rely on the NAG C Library because of the quality and accuracy the software gives to their work.

Containing over 1000 functions covering a wide range of mathematical and statistical areas, the NAG C Library functions are powerful, reliable and available on a wide range of systems commonly used for technical computing. NAG C Library Contents

Below is a list of some of the main numerical and statistical capabilities of the Library. Further details on the contents of the Library are included in the online manual.

- Optimization, including linear, quadratic, integer and nonlinear programming, least squares problems, constrained minimization and constrained quadratic programming for large sparse problems
- Ordinary and partial differential equations, and mesh generation
- Numerical integration
- Roots of nonlinear equations (including polynomials)
- Solution of dense, banded and sparse linear equations and eigenvalue problems
- Special functions
- Curve and surface fitting and interpolation
- Large scale eigen problems
- Large, sparse systems of linear equations
- Random number generation
- Simple calculations on statistical data
- Correlation and regression analysis
- Multivariate methods

6.2 Visualisation

One facet of software that is associated with C++ is the user interface and the really interesting part of the user interface for scientific software is the presentation of results. Text files are all very well for the testing phase but for wowing potential users and funders, exciting graphics is what are required! That is what this section is all about, libraries that can make producing great looking graphics from results easy(er).

6.2.1 The Visualization ToolKit—VTK

VTK (`http://www.vtk.org/`) is an open source, freely available software system for 3D computer graphics, image processing, and visualization used by thousands of researchers and developers

around the world. VTK consists of a C++ class library, and several interpreted interface layers including Tcl/Tk, Java, and Python.

VTK supports a wide variety of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods; and advanced modeling techniques such as implicit modelling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. In addition, dozens of imaging algorithms have been directly integrated to allow the user to mix 2D imaging / 3D graphics algorithms and data. The design and implementation of the library has been strongly influenced by object-oriented principles.

6.2.2 PLplot

PLplot (`http://plplot.sourceforge.net/`) is a cross-platform software package for creating scientific plots. To help accomplish that task it is organized as a core C library, language bindings for that library (including C, C++ and Fortran), and device drivers which control how the plots are presented in non-interactive and interactive plotting contexts.

The PLplot core library can be used to create standard x-y plots, semi-log plots, log-log plots, contour plots, 3D surface plots, mesh plots, bar charts and pie charts. Multiple graphs (of the same or different sizes) may be placed on a single page, and multiple pages are allowed for those device formats that support them.

6.2.3 PGPLOT

The PGPLOT Graphics Subroutine Library (`http://www.astro.caltech.edu/~tjp/pgplot/`) is a Fortran- or C-callable, device-independent graphics package for making simple scientific graphs. It is intended for making graphical images of publication quality with minimum effort on the part of the user. For most applications, the program can be device-independent, and the output can be directed to the appropriate device at run time.

The PGPLOT library consists of two major parts: a device-independent part and a set of device-dependent "device handler" subroutines for output on various terminals, image displays, dot-matrix printers, laser printers, and pen plotters. Common file formats supported include PostScript and GIF.

PGPLOT itself is written mostly in standard Fortran-77, with a few non-standard, system-dependent subroutines. PGPLOT subroutines can be called directly from a Fortran-77 or Fortran-90 program. A C binding library (cpgplot) and header file (cpgplot.h) are provided that allow PGPLOT to be called from a C or C++ program; the binding library handles conversion between C and Fortran argument-passing conventions.

6.3 Other Libraries

The aim here is to give brief details on useful libraries that a novice C/C++ developer might want to know about as he or she starts out.

6.3.1 GLib

GLib is the underlying C library for GTK+ (`http://www.gtk.org`) the Gimp ToolKit. GLib provides the fundamental algorithmic language constructs commonly duplicated in applications. This library has features including:

- Object and type system
- Main loop

- Dynamic loading of modules (i.e. plug-ins)
- Thread support
- Timer support
- Memory allocator
- Threaded Queues (synchronous and asynchronous)
- Lists (singly linked, doubly linked, double ended)
- Hash tables
- Arrays
- Trees (N-ary and binary balanced)
- String utilities and charset handling
- Lexical scanner and XML parser
- Base64 (encoding & decoding)

6.3.2 Boost

From the website `http://www.boost.org/`.

> Boost provides free peer-reviewed portable C++ source libraries.
>
> We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use.
>
> We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) as a step toward becoming part of a future C++ Standard. More Boost libraries are proposed for the upcoming TR2.

The idea is to boost programmer productivity.

References

[1] D.J. Worth and C. Greenough, A Survey of Software Tools for Computational Science, RAL Technical Report RAL-TR-2006-011, 2006.

[2] The GNU Scientific Library, `http://www.gnu.org/software/gsl/`.

[3] SciMath from Advanced Scientific Applications Inc., `http://www.scimath.com/`.

[4] LAPACK++, `http://sourceforge.net/projects/lapackpp/`.

[5] CHARM++, A machine independent parallel programming system, `http://charm.cs.uiuc.edu/research/charm/`.

[6] T. Veldhuizen, Scientific Computing: C++ Versus Fortran: C++ has more than caught up, Dr. Dobb's Journal of Software Tools, **22**, 11, pp 34, 36–38, 91, Nov 1997.

[7] T. Veldhuizen, Techniques for Scientific C++, Indiana University Computer Science Technical Report 542 Version 0.4, August 2000. Also available at `http://ubiety.uwaterloo.ca/~tveldhui/papers/techniques/techniques.html`.

[8] The Unified Modeling Language, `http://en.wikipedia.org/wiki/Unified\_Modeling\_Language`.

[9] T.J. McCabe, A complexity measure, *IEEE Trans. Software Eng.*, **SE2**, no 4, pp 308–320, 1976.

[10] S.R. Chidamber and C.F. Kemerer. A Metrics suite for Object Oriented design. M.I.T. Sloan School of Management E53-315. 1993.

[11] D. Kafura, and S.M. Henry, Software Quality Metrics Based on Interconnectivity, *Journal of Systems and Software*, **2**, pp. 121–131, 1982.

[12] B.A. Nejmeh, NPATH: a measure of execution path complexity and its applications, *Commun, ACM*, **31**, no 2, pp 188–200, 1988.

[13] A Survey of Software Testing Tools for Computational Science, L.S. Chin, D.J. Worth, and C. Greenough, RAL Technical Report RAL-TR-2007-010, July 2007.

[14] Cygwin, a Linux-like environment for Windows, `http://www.cygwin.com/`.

[15] MinGW, Minimalist GNU for Windows, `http://www.mingw.com/`.

[16] Link Exchange for the Technical Computing Community, C/C++ Section `http://www.mathtools.net/C_C__/index.html`.

[17] The Object-Oriented Numerics Page, `http://www.oonumerics.org/oon/`.