

Open Source Messaging Application Development: Building and Extending Gaim

SEAN EGAN

Open Source Messaging Application Development: Building and Extending Gaim

Copyright © 2005 by Sean Egan

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-467-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Jason Gilmore

Technical Reviewer: Nathan Walp

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Associate Publisher: Grace Wong

Project Manager: Beth Christmas

Copy Edit Manager: Nicole LeClerc

Copy Editor: Candace English

Production Manager: Kari Brooks-Copony

Production Editor: Kelly Winkist

Compositor: Susan Glinert and Wordstop Technologies Pvt. Ltd., Chennai

Proofreader: Linda Seifert

Indexer: Broccoli Information Services

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders@springer-ny.com, or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit <http://springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.



Development Tools

The knowledge presented in this book will teach you how to create your own desktop applications using open source technologies. That knowledge, however, is useless without the development tools needed to put it to use. The open source tools introduced in this chapter are among those most commonly used for software development.

Chapter 1 guided you through the process of installing most of the software that will be explained here. If you are a UNIX user, they were most likely already installed; Windows users installed the software using Cygwin. In this chapter, I will explain how to use it.

Editors

It would be pretty difficult to write effective code without some sort of editing application. Capable editors not only provide a workspace for writing code, but also a variety of tools useful for quickly and efficiently managing it. There is a huge variety of editors out there, and if you have one you're already comfortable with, you can go on using that. However, some editors are simply less suitable for writing code than others. For instance, most people can be far more productive with vi or Emacs than with Nano or Windows Notepad.

There are two major type editors in the UNIX world: Emacs and vi. Most other editors borrow features from one or both of these. The two are very different, and a large part of UNIX humor is the so-called “religious war” between advocates of each. Emacs is applauded for its power and versatility and criticized for being large and slow. vi is liked for being small and fast, but is criticized for its confusing user interface.

Both vi and Emacs, however, are very well suited for editing code, and most UNIX users may have a preference for one over the other but are familiar with both. A few things make these editors considerably better suited for writing code than other editors:

- Both editors support *syntax highlighting*, which colors bits of code depending on their syntactical function. For instance, comments may be orange and variable names blue.
- Both editors understand commonly used indentation styles. After hitting the Enter key, the cursor will probably have appropriately indented the next line.

- Both editors have easily accessible keyboard shortcuts for things particularly useful to coders. Simpler editors require you select text to delete it; code editors provide special commands for deleting the current line, deleting the current word, and deleting from the cursor to the end of the line, for instance.
- Both editors can be integrated with the GNU build system, to be discussed throughout this chapter.
- Gaim developers will make fun of you if you use an editor like Pico.

Personally, I use an Emacs editor (specifically XEmacs) for most of my coding, and a vi editor (specifically Vim) for smaller jobs, such as fixing a syntax error if my compile failed. I appreciate Emacs's more familiar user interface (especially the way it handles editing multiple files), but don't like to wait for it to load when I need to change only one line of code.

I'll skim the surface of both editors here and provide enough information to give you a passable knowledge of how to use each.

Emacs

Emacs is my preferred editor for working on large projects due to its more familiar interface. In Emacs, commands are entered by using combinations of the Ctrl, Alt, and other ordinary keys; the way commands are entered is the most notable difference between Emacs and vi. This way is probably most familiar to you, as most editors use it. If you didn't install Emacs from Cygwin, you can download a native Windows version from <http://www.xemacs.org>.

Features

I mentioned that Emacs handles multiple files better, which I think makes it better for larger projects. The version of Emacs I use, XEmacs, displays files in a tabbed interface, shown in Figure 3-1, which I find makes it easier to switch between two open files than the vi approach.

Also seen in Figure 3-1 is the game Tetris, which is part of Emacs and a source of criticisms that it's too "bloated." Emacs has always had a powerful Lisp plug-in architecture, and over time many plug-ins have been developed. From within Emacs you can use a virtual psychiatrist, a calendar, a mail reader, and even an AIM client.

Emacs was one of the first programs to come out of the GNU project, so it's well integrated with the GNU development tools. You can compile and debug your applications from within Emacs. Despite the powerful features it contains, though, I tend to use it only for simple editing.

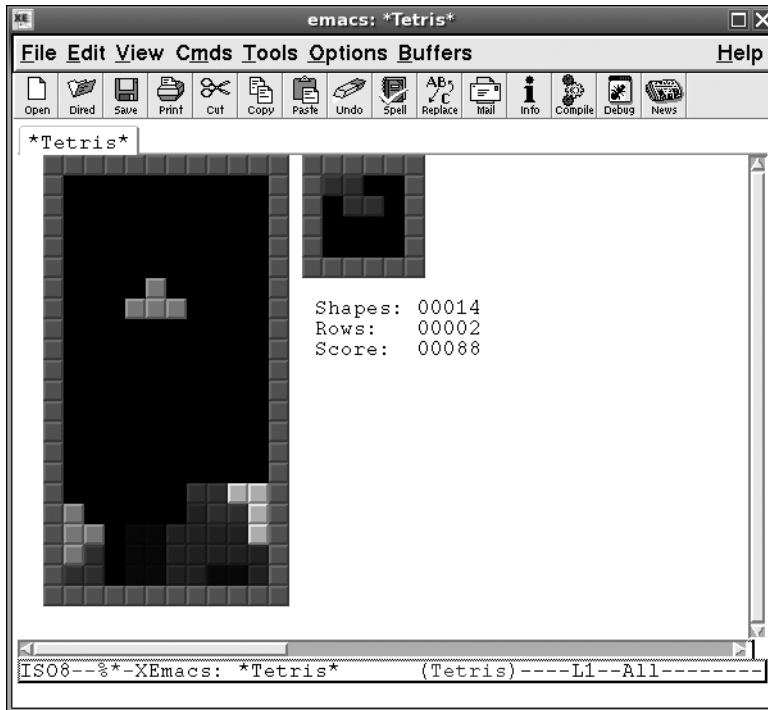


Figure 3-1. The Emacs editors (XEmacs is shown here) are often considered “bloated” due to features like this Tetris game.

Usage

Every keystroke in Emacs is a command. For most of these keystrokes, the command is self-insert-command. This is the command that causes Emacs to insert a character into the current document. When you type the letter I, the self-insert-command is called and an I is inserted into the document. For other commands, though, Emacs makes heavy use of two modifier keys called Control and Meta.

These two keys are typically abbreviated C and M, so when you read C-x, this means “hold down Control, and press the A key.” Most PC users have a Control key on their keyboard, but not a Meta key. On these machines, the Meta key usually maps to the Alt key, so M-f means “hold down Alt, and press F.”

Some commands require more than one keystroke. The first keystroke is called a *prefix command*, and waits for a second keystroke. Functions with the same prefix command are typically similar in function. For instance, C-h is used by commands that provide help. Try C-h i (which is different from C-h C-i) to read the Emacs manual.

Common Commands

XEmacs also includes a menu and a toolbar, so it's not necessary that you remember all of its many cryptic commands. However, many people prefer to keep their hands on the keyboard at all times and prefer a keyboard command to a menu or toolbar button.

The C-x prefix command includes most of the functions for managing various files and documents. Table 3-1 shows some of the more useful commands.

Table 3-1. *Emacs Commands for Manipulating Files*

Command	Description
C-x C-f	Opens a file for editing
C-x C-s	Saves a file
C-x C-k	Closes a file
C-x C-c	Exits Emacs

Once you have your file loaded, you can simply type into it; each keystroke will insert its character. Usually you can also navigate the document with your keyboard's arrow keys. However, when coding it's often useful to have finer control over navigation. Table 3-2 provides some useful commands for getting from one place in your file to another.

Table 3-2. *Emacs Commands for Navigating Within a File*

Command	Description
C-f	Moves forward one character
C-b	Moves backward one character
M-f	Moves forward one word
M-b	Moves backward one word
C-a	Moves to the beginning of the line
C-e	Moves to the end of the line
C-p	Moves up one line
C-n	Moves down one line
M-<	Moves to the beginning of the file
M->	Moves to the end of the file
C-M-a	Moves to the beginning of the current function
C-M-e	Moves to the end of the current function

Of course, often you don't know precisely where within a file the code you're looking for is useful. Other times, you need to replace all instances of one string with another, perhaps to rename a variable or function. For these tasks, you should use Emacs's search and replace features.

Emacs features *incremental search*, which means that as you type the phrase you're searching for, Emacs goes to the first instance of what you've written already. This is good because you'll rarely need to type the entire phrase you're searching for. Emacs will locate it based on the first few characters. Table 3-3 shows Emacs's search and replace functions.

Table 3-3. *Emacs Commands for Searching for and Replacing Text*

Command	Description
C-s	Incremental search.
M-% (Alt-Shift-5)	Find and replace; Emacs will prompt for the text to search for and the text to replace it with.

Lastly, once you've found text, you may need to delete it or move it. Depending on how your Emacs is configured, you should be able to use the Backspace and Delete keys as you would expect them to work, but just as Emacs offers commands to navigate based on lines and words, it also allows you to delete elements larger than a character.

When you delete something this way, it gets saved to Emacs's equivalent of a clipboard. You can then paste the deleted text somewhere else in the text. Emacs offers no other special commands for copying and pasting. In fact, to copy text, I just delete it and then immediately paste it. Depending on your system, Emacs may integrate with the native clipboard. Table 3-4 shows the commands used to delete text.

Tip The standard way to copy text to the X Window System is to select it. Any selected text is implicitly in the clipboard. Then use the middle mouse button (often the scroll wheel) to paste.

Table 3-4. *Emacs Commands for Deleting, Copying, and Pasting Text*

Command	Description
M-d	Deletes from the cursor to the end of the current word
M-DEL	Deletes from the beginning of the current word to the cursor
C-k	Deletes from the cursor to the end of the current line
C-u C-0 C-k	Deletes from the beginning of the current line to the cursor
C-y	Pastes previously deleted text at the cursor

These functions merely scratch the surface of what Emacs is capable of, but they make up nearly 100% of what I use it for. You can read Emacs's own help documentation via C-h i.

vi

When I have only one file that needs editing, I'll often use Vim, a vi editor seen in Figure 3-2. Because of Emacs's bulk, it loads much more slowly than vim does. However, I prefer the way XEmacs shows me currently open files in tabs and that Emacs is not modal, as vi is.

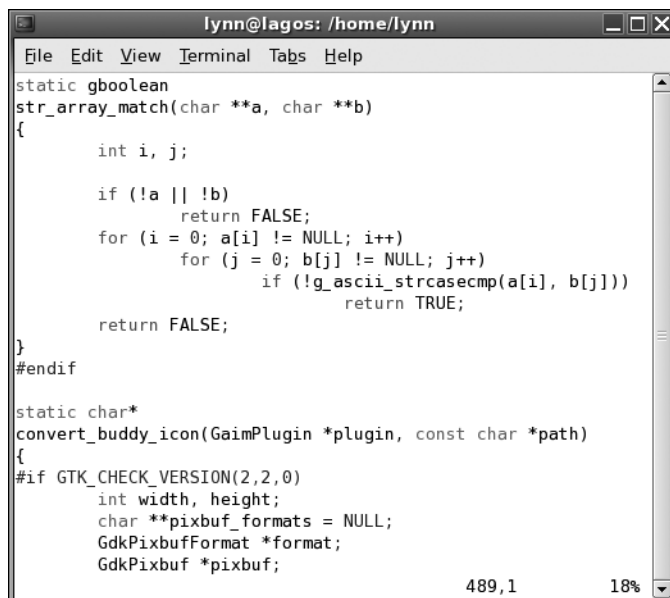


Figure 3-2. *The Vim editor*

Modes

vi is modal, meaning it has two different modes: insert and command. When in insert mode, everything you type is inserted in the current document. When in command mode, everything you type is a command. Using vi requires you to constantly move between the two modes.

vi fans say this a good thing, that they can always keep their fingers on the home keys of their keyboard without needing to move them to hit the Control or Meta key. I, personally, would prefer to move my hands occasionally to avoid the confusion of forgetting what mode I'm in, causing text I attempt to insert to issue commands. The opposite happens too, and I've written code that strangely contains vi commands.

vi starts in command mode. To enter insert mode, press the I key. It will move you to insert mode; you can tell this because the bottom line of the screen will read -- INSERT --, as seen in Figure 3-3. To return to command mode, hit the Escape key.


```

gboolean is_auto_away;      /**< Whether or not it's auto-away. */

gboolean wants_to_die;      /**< Wants to Die state. This is set
                             when the user chooses to log out,
                             or when the protocol is
                             disconnected and should not be
                             automatically reconnected
                             (incorrect password, etc.) */
guint disconnect_timeout;   /**< Timer used for nasty stack tricks */
};

#ifdef __cplusplus
extern "C" {
#endif

/***** @name Connection API *****/
/***** @name Connection API *****/
/*@{*/

/**
 * This function should only be called by gaim_account_connect()
 * in account.c. If you're trying to sign on an account, use that
 * function instead.
 *
 * Creates a connection to the specified account and either connects
 * or attempts to register a new account. If you are logging in,
 * the connection uses the current active status for this account.
 * So if you want to sign on as "away," for example, you need to
 *
 * -- INSERT --

```

Figure 3-3. *vim in insert mode, as evidenced by the bottom line on the screen*

Commands

The variant of vi I use, Vim, is a console-based application, and like Emacs, native versions exist for most any platform you would want it for. There are also GUI versions, such as GVim (which run on X or Windows). They include menus and toolbars, but because not waiting for a GUI to load is the major advantage for me in using Vim over XEmacs, I enter all my commands with the keyboard.

vi commands do not use modifier keys such as Control or Meta as Emacs does. vi commands are entered in the command line and are usually short words or just single characters. Whereas Emacs commands that deal with file management start with C-x, the same commands in vi start with a colon. These commands are listed in Table 3-5.

Table 3-5. *vi Commands for Manipulating Files*

Command	Description
:r	Opens a file.
:w	Saves a file.
:q	Quits vi if there are no changes to save. If there are, vi will not quit and will instead warn about unsaved changes; you'll need to use one of the two following commands.
:wq	Saves and quits.
:q!	Quits without saving.

vi also allows you to navigate the file with cursor keys, but most people recommend navigating with the keys shown in first four rows of Table 3-6 so that you can keep your hands in place. Like Emacs, vi can navigate through a file more than one character at a time. Some of the commands to do this are found in Table 3-6.

Table 3-6. *vi Commands for Navigating Within a File*

Command	Description
h	Moves the cursor left one column.
j	Moves the cursor down one line.
k	Moves the cursor up one line.
l	Moves the cursor right one column
w	Moves forward one word.
b	Moves backward one word.
L	Moves to the bottom of the screen.
H	Moves to the top of the screen.
O	Moves to the beginning of the current line.
\$	Moves to the end of the current line.
-	Moves to the first non-whitespace character of the current line; useful for skipping indentation.
<num>	Goes to the line specified in <num>. For instance, to move to line 44, do :44.

Also like Emacs, vi allows you to delete segments of text as well as just characters. These commands build on top of the navigation commands. To delete from the cursor to another location, type *d* and then the command to move the cursor to that location. To delete until the end of the current line, for instance, you would do *d\$*. As a special case, *dd* deletes the entire current line. As in Emacs, deleted text is automatically “cut” to the clipboard. To insert text, use the *p* command.

By default, vi does not do incremental search; you must search for an entire phrase (of course, you could just search for the first few characters of your search term, but you will have to skip through nonmatches manually). Table 3-7 shows the vi commands for searching and replacing text.

Table 3-7. *vi Commands for Searching and Replacing Text*

Command	Description
<code>/<word></code>	Searches for <code><word></code> . To search for “twinkies,” do <code>/twinkies</code> .
<code>?<word></code>	Searches for <code><word></code> backwards through the document.
<code>n</code>	Repeats the previous search.
<code>:s/<search string>/<replacement string>/g</code>	Replaces elements of <code><search string></code> with <code><replacement string></code> . The <code>g</code> means “global,” and causes all instances of <code><search string></code> to be replaced. To do it only to the first instance, leave off the <code>g</code> .

Note Many UNIX developers are so familiar with `vi`’s search and replace syntax that they use it in e-mails, IRC, and IM. If someone IMs you `s/apples/bananas/g`, he means “replace ‘apples’ with ‘bananas’ in what was just said.”

As you can see, Emacs and `vi` offer essentially the same features, but with considerably different user interfaces. An alternate user interface that may be more familiar to Windows developers is the Integrated Development Environment, or IDE, which integrates the editor with other development tools.

IDEs

Windows developers are probably familiar with Integrated Development Environments such as Microsoft Visual Studio or Microsoft Visual Basic. IDEs integrate the editor, compiler, and debugger into a single program. Due largely to tradition, these tools are rarely used on UNIX. Instead, developers use each program in the GNU build system separately.

However, various free software IDEs for C programming are available for Windows and UNIX. I’ll list a few of them here.

Anjuta

Anjuta, pictured in Figure 3-4, is an IDE for UNIX’s GNOME desktop environment. As such, it integrates well with the GNOME environment (see Chapter 5 for more on GNOME). In addition to a fully functional editor, Anjuta integrates with GCC (the GNU Compiler Collection) and GDB (GNU Debugger), which I’ll cover later in this chapter.

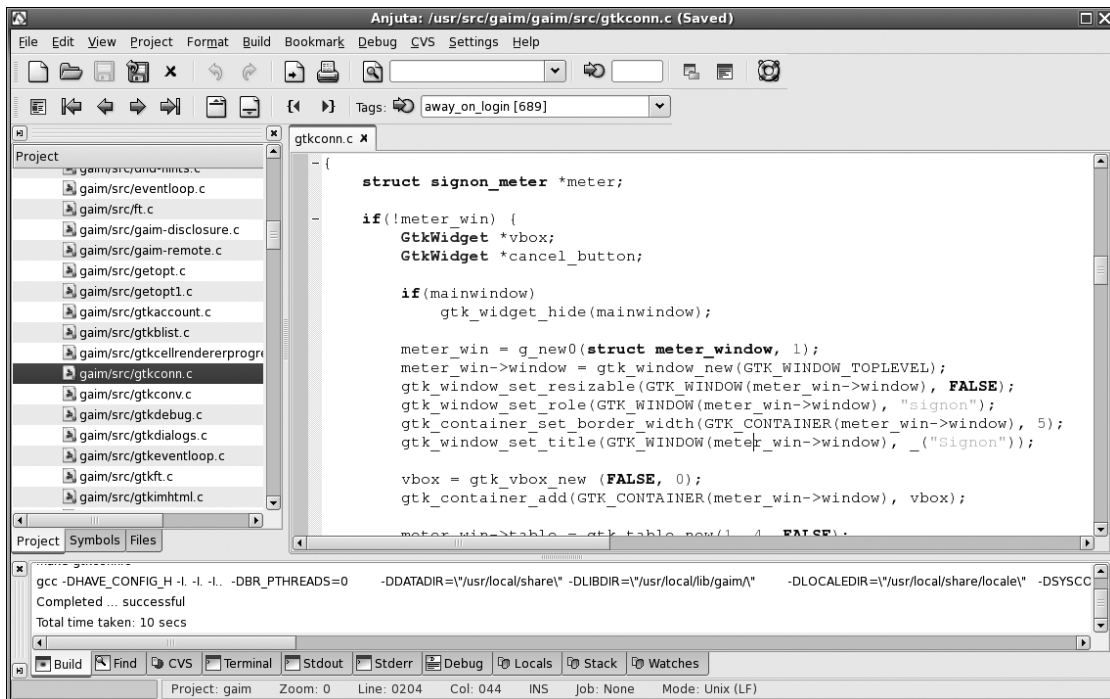


Figure 3-4. *The Anjuta IDE*

Because it's developed for GNOME, it is particularly well suited for developing GNOME applications. When you create a new project in Anjuta, it will automatically create a build environment that links to the required GNOME libraries. It will also create skeleton source code to create a window. You can then create a GUI using Glade, GNOME's WYSIWYG interface builder, and edit the code to suit your application. This certainly is less work than writing everything manually. However, Gaim and other UNIX applications typically do not use IDEs or interface builders, and prefer that everything is written manually.

You can download Anjuta from <http://anjuta.sourceforge.net/>.

KDevelop

KDevelop is to the KDE environment as Anjuta is to the GNOME environment. It offers the normal editor, compiler, and debugger integration you'd expect from any IDE, but it also offers functionality specific to developing KDE applications. You can see KDevelop in Figure 3-5.

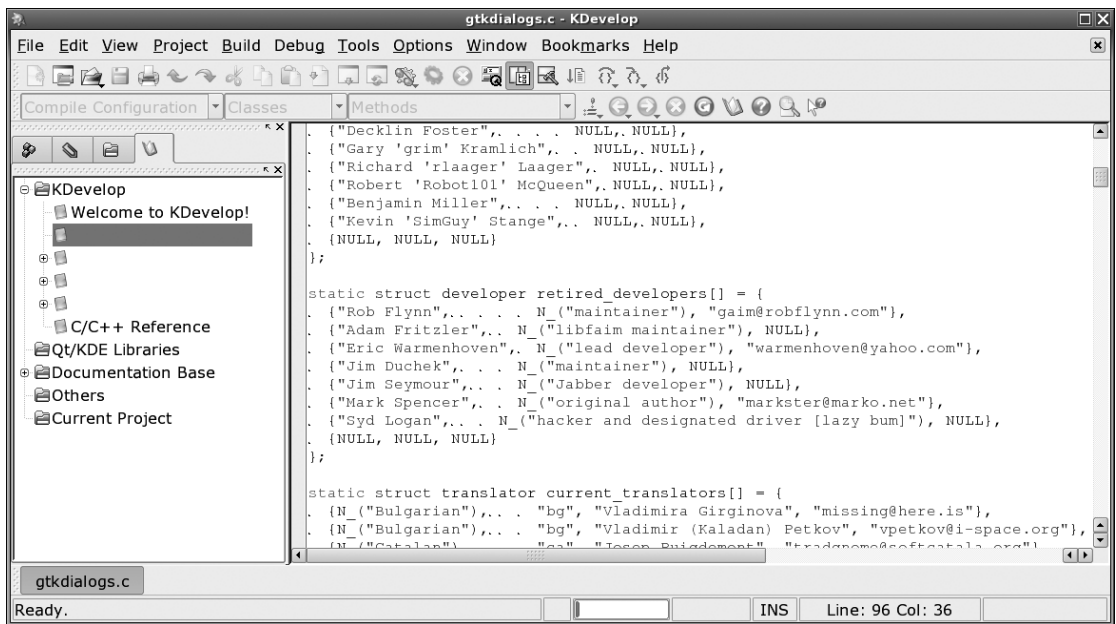


Figure 3-5. KDevelop is an IDE specially suited for creating KDE applications.

KDE applications use Trolltech's Qt (pronounced "cute") toolkit, whereas Gaim uses GTK+. Therefore, many of KDevelop's features are specifically tailored for Qt and not appropriate for developing Gaim or other GTK+ applications. KDevelop does, however, have some support for GNOME applications and is the most mature UNIX IDE for development in C.

You can download KDevelop from <http://www.kdevelop.org>.

Dev-C++

Dev-C++, seen in Figure 3-6, is an IDE for Windows. It uses MinGW, the Windows version of GCC capable of creating Windows-native applications, and the Windows port of GDB, the GNU debugger.

Like KDevelop and Anjuta, Dev-C++ is modeled after Microsoft Visual Studio, which makes it easy to use for Windows developers coming from those environments. Also like the others, Dev-C++ contains features specifically useful for developing for its target platform, Windows.

Dev-C++ is available under the GPL license from Bloodshed Software at <http://www.bloodshed.net>.

Using an IDE will obscure many of the details of the build and debugging process I will discuss in the next few sections. Each of the IDEs listed here are just front ends to these tools, however, so it is still important to understand what they are and the basics of how they work.

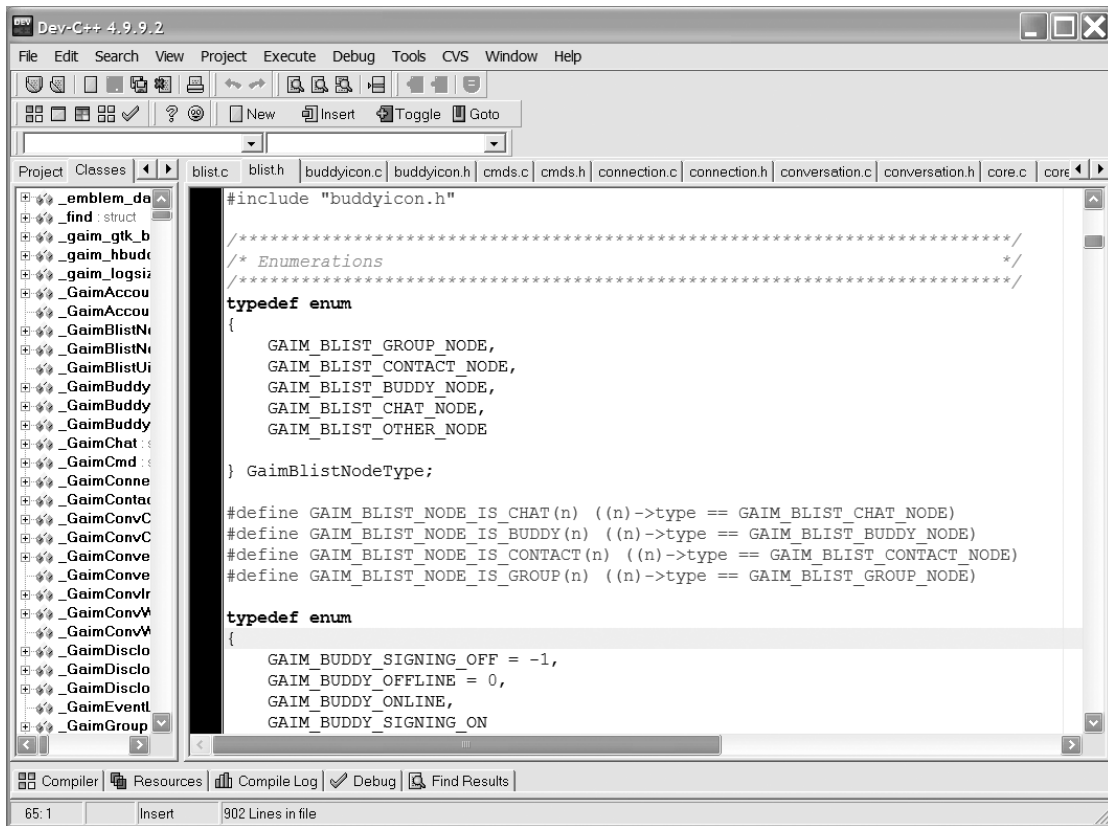


Figure 3-6. Dev-C++, an IDE for Microsoft Windows

GCC

GCC is the commonly used acronym for the GNU Compiler Collection (<http://gcc.gnu.org/>). It was originally the GNU C Compiler, but that name is no longer accurate as GCC is now capable of compiling source in C, C++, Objective-C, Fortran, Java, and Ada. It is very portable; GCC is capable of compiling source for nearly every platform imaginable, and because GCC is written in C and compiled by itself, GCC is likewise available on most platforms. It is the compiler used by Gaim, almost all free software, and other non-free software.

GCC is the crux of the GNU build system. The GNU build system, used to compile nearly all free and open-source software, consists of GCC and GNU Autotools, discussed later this chapter.

Note MinGW is a Windows version of GCC; everything I say about GCC here is true of MinGW as well.

The compiler is the lowest-level tool in the development toolchain. Its task is straightforward; it reads source files and compiles them to binary executable files. Because it's so low-level, developers rarely invoke GCC directly but allow it to be invoked by other tools. However, to properly configure these tools for your project, you should know the basics of how to use GCC and how GCC works. The process of compiling source to executable happens in two main, yet distinct, phases: compilation and linking.

How a Program Is Built

Most C applications of any considerable size take up more than one file. Each of these files is first compiled individually into an object file with the same base name as the source file, but with an `.o` extension. Thus, when you compile `account.c`, it creates a file called `account.o`. This file contains all the executable machine code for the source code contained within `account.c`, as well as the location of variables and functions.

The linker takes a number of object files and compiles them to a single executable. The linker matches references to a function or variable in one object file to the correct location of that function or variable in another object file. If `account.c` calls the `gaim_connection_new()` function from `connection.c`, the linker will find `gaim_connection_new()` from `connection.o` and match it to the call from `account.o`.

It's possible to perform compilation and linking on the same command line. This is the default behavior of GCC when no arguments are given. However, that would require recompiling the entire program every time a single source file changed. Therefore, one typically compiles each source file as it's changed, a task made simple with the `make` tool, discussed later. `make` is the tool that invokes GCC with the appropriate arguments.

Invoking GCC

GCC uses command-line arguments to affect the compilation process. These options can change a wide array of behaviors, from how the language is parsed, to what language features are enabled, to how well optimized the resulting executable should be. The most basic command-line options, though, merely specify what files the compiler should compile, link, or both compile and link.

Compiling and Linking

The default behavior for GCC when passed no arguments other than source files is to compile all the provided source files and link them into an executable called `a.out`. To create an executable with a different name, use the `-o` command-line option:

```
gcc -o exampleprogram *.c
```

This command compiles all the C source files in the current directory and links them to an executable called `exampleprogram`. If you've already compiled the source files, you can link the object files by providing only a list of them to GCC, using the `-o` option if desired:

```
gcc -o exampleprogram *.o
```

This command creates the `exampleprogram` executable from the object files in the current directory. It will not recompile source files, even if they've changed. To do that, specify source files to compile with the `-c` option:

```
gcc -c *.c
```

This will create object files for each source file. Usually, though, each file is compiled on a separate invocation of GCC. The typical process in compiling multiple source files into a single binary executable would look like this:

```
gcc -c file1.c
gcc -c file2.c
gcc -c file3.c
gcc -o exampleprogram file1.o file2.o file 3.o
```

The first three commands create object files for each source and the fourth links them to an executable called `exampleprogram`. This process would work fine for small projects with no external dependencies, but once a project depends on some external library, GCC must be told where that library is located.

Dependencies

Your application will likely compile and link against a number of libraries, such as GTK+. Libraries your application uses are called *dependencies* and need to be located in both the compilation and linking processes. During compilation, the compiler needs to be given the location of `.h` header files that are `#included` by the project. Without them, the compilation would fail: the headers would not be located and the compiler would not know about the data types and functions provided by the library.

During linking, the compiler needs to know the location of each library on disk. These libraries are typically “dynamically linked,” meaning that the code in your application isn’t actually linked with the code in its libraries until right before the program runs. Nonetheless, the linker needs to know what libraries the executable will be linked against and where those libraries are located. The location of the library files and header files are unrelated to each other; they must be determined and specified to GCC separately through a series of command-line options.

To provide the compiler with a list of locations to fetch header files from, use the `-I` command-line option, followed by a path to the directory. You will need one `-I` for each directory you compile against. Gaim compiles against a large number of libraries, so each invocation of GCC has nearly 20 `-I` options. Fortunately, the task of maintaining these options is handled by GNU Autotools, discussed later this chapter.

The linker takes two different command-line options for specifying which libraries to link against. The first option, `-l`, specifies which libraries to link against. The name you specify isn’t a file name, but a library name. It will correspond to a file with the specified name prefixed with `lib` and postfixed with `.a`. Thus, when the build environment gives an `-lglib-2.0` option to GCC, it links against a file named `libglib-2.0.a`.

This file, `libglib-2.0.a`, is located in a directory specified by the `-L` command. `-L` works just like `-I`. Each directory is specified with its own argument. Often, each library will require both an `-l` and an `-L` argument, if the library file is not found in a standard location.

Other Options

Other important options to GCC include `-D`, which defines a preprocessor statement during compilation. This allows the programmer to selectively choose what to compile at compile-time. One common use of this is to allow additional debugging support by providing `-DDEBUG` to GCC. Then, within your code you could write

```
#if DEBUG
printf("%s\n", debug_msg);
#endif
```

The `printf()` line will be compiled in only if `DEBUG` is defined by the `-D` argument.

`-W` provides another useful set of compiler options. These allow you to specify how compile warnings are handled. Compile warnings are bits of code that are not illegal according to the specification, but are still most likely wrong. If your code has warnings in it, it will compile, but it may have bugs. `-Wall` will activate all warnings; the default is to show only the most serious ones. `-Werror` causes GCC to treat warnings as errors; the process exits as a failure.

Finally, to enable debugging information, provide `-g` GCC while linking. This will allow you to usefully run the GNU debugger, GDB, on the resulting executable. I will discuss GDB later this chapter.

All these command-line options plus the sheer number of times GCC is invoked during the process of building a project makes it quite unwieldy to perform the task by hand. The `make` utility is the next step in the build process; it allows you to describe the build process in a *makefile*, which `make` uses to provide the appropriate command options to GCC.

make

`make` is an extremely useful part of the build process. By creating a file called *Makefile*, you describe to `make` exactly how to build your application. `make` can then intelligently determine how to build your application without necessarily rebuilding the entire application.

If you build *Gaim* and then change a single source file, only that source file needs to be recompiled. The other object files are up-to-date. All that needs to be done is to compile that one source file and link the resulting object file with the other, older object files. By creating a proper *makefile*, you can specify this behavior by specifying separate *rules* for each sub-task in building your application.

Makefile Rules

A *makefile* is essentially a set of rules. Each rule typically describes how to create a given file. A rule contains the name of the file to create, prerequisites that the file depends on, and the process for creating the file:

```
exampleprogram : file1.o file2.o file3.o
    gcc -o exampleprogram file1.o file2.o file3.o
```

This example is the rule for creating a file called `exampleprogram`. The target file is located at the beginning of the first line. Following the target is a colon and then a list of prerequisites. In order to build an up-to-date `exampleprogram`, `make` must first build an up-to-date `file1.o`, `file2.o`, and `file3.o`. If `exampleprogram` does not exist, or it does exist but happens to be older

than `file1.o`, `file2.o`, or `file3.o`, `make` will execute the command on the next line. If `exampleprogram` is newer than its prerequisites (including the prerequisites of its prerequisites), `make` realizes that `exampleprogram` is also current, and doesn't need to execute the command.

The line the command starts on begins with a tab followed by the command that will be given to the shell to execute. In this case, it tells GCC to link the object files to an executable called `exampleprogram`, using the syntax seen in the section on GCC. If all the `.o` files are up-to-date, it will issue that command and link them. However, if an `.o` file needs to be updated, `make` will figure out how to do it with its own makefile rule:

```
file1.o : file1.c
    gcc -o file1.o -c file1.c
file2.o : file2.c file2.h
    gcc -o file2.o -c file2.c
file3.o : file3.c
    gcc -o file3.o -c file2.c
```

Each object file gets its own makefile rule. The dependencies include the source file as well as any headers in the rule. Because the headers are included into the source file, the source files need to be recompiled if a header changes. Fortunately, the GNU Autotools will automatically keep track of which header files should be used as prerequisites.

Implicit Rules

Notice that the rules for the three `.o` targets above are essentially the same, differing only in the filenames. With a large project, like Gaim, with over 100 source files, maintaining a separate, nearly identical build target for each would be inconvenient and error-prone. Because of this problem, `make` allows you to create *implicit rules*. These rules don't describe how to build a specific file, but rather a rule that describes generally, for instance, how to create an `.o` from a `.c` file. Gaim's Windows makefile includes a rule like this:

```
%.o: %.c
    $(CC) $(CFLAGS) $(INCLUDE_PATHS) $(DEFINES) -c $< -o $@
```

This allows you to avoid excessively long, largely redundant, difficult-to-maintain makefiles. If `make` is told to build an `.o` file, it will first look to see if that specific file has a rule in the makefile. If not, it will attempt to match it to an implicit rule. The `%` in `%.o` matches to any word; `%.o` matches any file ending with `.o`. In the prerequisite section, the `%` matches the same word, so if `account.o` were matched to `%.o`, it would depend on `account.c`.

The command looks different from the command shown earlier in this section. The `$` symbol is used to start variables. Every term starting with a `$` is a variable. The last two, `$<` and `$@`, are automatic variables, which are evaluated to the input file and the output file, respectively. The first few, `$(CC)`, `$(CFLAGS)`, `$(INCLUDES)`, etc., are declared elsewhere in the makefile.

Variables

Because certain aspects of the build environment are very dynamic and one build environment may vary greatly from another, it needs to be easy to change certain aspects of the build process. You can accomplish this by declaring anything likely to change as variables, listed at the top of the file.

Recall the first example of a makefile rule I provided:

```
exampleprogram : file1.o file2.o file3.o
    gcc $(LIBS) -o exampleprogram file1.o file2.o file3.o
```

Notice how the list of object files is duplicated. Because adding, removing, or renaming source files is not uncommon during the development process, this list will likely change often. Every time it's changed, this makefile rule needs to be updated in two places (in both lines of this makefile rule). It might be easy to forget to update both, resulting in a broken makefile.

It would be easier, then, if we used a variable called `$(OBJECTS)` to represent the list of all the object files in the program. Then, the makefile rule would read

```
exampleprogram : $(OBJECTS)
    gcc $(LIBS) -o exampleprogram $(OBJECTS)
```

The variable declaration at the top of the makefile is the only thing that would require editing:

```
OBJECTS = file1.o \
          file2.o \
          file3.o
```

Note that backslashes are used to split the declaration among several lines to make it easier to read. Other things are useful as makefile variables as well. The implicit rule example uses `$(CC)`, `$(CFLAGS)`, `$(INCLUDE_PATHS)`, `$(LIBS)`, and `$(DEFINES)`. These variables correspond to the compiler command (`gcc`), compiler flags (`-g -Wall`, perhaps), header file locations (`-I/usr/include/gtk+-2.0 -I/usr/include/glib-2.0`, etc.), and defined preprocessor statements (`-DDEBUG`), respectively.

For compiling GTK+ applications, you will want to set `$(CFLAGS)` and `$(LIBS)`, to point to GTK+'s header files and shared object files, respectively. A convenient way to do this is to invoke `pkg-config`, which I'll cover in greater detail in the Autotools section of this chapter. Typical variable definitions might look like this:

```
CFLAGS=`pkg-config --cflags gtk+-2.0`
LIBS=`pkg-config --libs gtk+-2.0`
```

Multiple Directories

Another notable point is that for projects with more than one directory, `make` can be used recursively. The command for a given makefile rule can run `make` in another directory. This is useful because most source trees contain multiple directories for different types of files: Gaim includes source code to be compiled in `src/` and `plugins/`. The top-level makefile—the makefile in the top-level directory—just dispatches `make` commands to its subdirectories.

Because no files are actually created in the top-level directory, Gaim's makefile uses a phony target called `all`, which builds the entire project. Several of these phony targets are fairly standard as they're created by GNU Autotools, discussed later in this chapter. Other common targets include `clean`, which deletes all of the compiled binaries, `install`, which installs the resulting executables in the proper locations, and `dist`, which creates a release tarball. A typical `all` target would look like this:

```
all:
    $(MAKE) -C src/
    $(MAKE) -C plugins/
    $(MAKE) -C pixmaps/
```

This rule would merely run `make` in each of the subdirectories provided. The person compiling this project would then just run

```
$ make all
```

to execute this target and build the application. If `all` is the first target defined in the makefile (which is usually the case), you just run

```
$ make
```

Running `make` is easy to run; most of the time (because of the behavior of Autotools), `all` is the first target defined, so you just run `make` and it builds the `all` target. However, you can also build any target you want on the command line. In the context of Gaim, this is particularly useful for compiling your own plug-ins. The makefile in Gaim's `plugin/` directory contains an implicit rule for creating a plug-in from a `.c` file. To compile `myplugin.c` into `myplugin.so` (or whichever the correct file extension is for plug-ins on your system), put it in `plugins/`, and from that directory run the following:

```
make myplugin.so
```

This will match the implicit rule for building `.so` files from `.c` files, and properly build your plug-in.

This is also useful when building the phony targets specified earlier. After building Gaim with just plain `make`, you need to build the `install` target. This is done just by running

```
make install
```

from the top-level directory.

Another useful command-line option to `make` is `-f`, which allows you to specify what file to use as the makefile. By default, `make` will look for a file called `Makefile`, and you don't need to specify this file manually. For a project like Gaim, however, building on Windows is vastly different from building on UNIX. Gaim, therefore, has two sets of makefiles. When building on Windows, you must specify to use the Windows makefile (called `Makefile.mingw`):

```
make -f Makefile.mingw
```

`make` is very useful, as it keeps you from manually invoking GCC hundreds of times. However, creating and maintaining a makefile, as you can tell, isn't very easy. Also, because one system varies greatly from another, a makefile would need to be edited for every new machine it was compiled on. Header file locations, for instance, can be almost everywhere.

Fortunately, the GNU Autotools exist so that users do not have to manually edit the makefile themselves. Autotools creates a shell script called `configure` that automatically determines what variables need to be set and generates a set of makefiles appropriately.

Autotools

When you know beforehand the exact configuration of the build environment, it is easy to create a working makefile for it. This is the case for Gaim's Windows build. In Chapter 1, I explained that in Windows each library is installed to a very specific location. Gaim's `Makefile.mingw`, used for building on Windows, is written assuming those locations. If you're developing solely on Windows, you can skip this section on Autotools.

Another time you can manually write an accurate makefile is if you're developing only on your own computer. You know the specifics about your own system. You know where each library is installed, and you can write your makefile accordingly. However, once you want to distribute your source tree to other people, you need to expect a huge variety of build configurations. GNU Autotools exists to create accurate makefiles from generic, higher-level descriptions of the build process. This allows makefiles to be built on almost any system imaginable.

Autotools consists of three utilities: Autoconf, Automake, and Libtool. These three applications work together to create a configure script that will generate relevant makefiles for a given system when invoked.

Autoconf creates the configure script. It uses a file called `configure.ac` as input. `configure.ac` specifies the variables in the build environment. Automake creates files called `Makefile.in`. These files are similar to makefiles in that they explain how the program is built. However, `Makefile.in` does it in a more general way, such that the configure script can process it with its own machine-specific information to create pertinent makefiles. Libtool is used to abstract the details involved in creating shared libraries. Because this book is about creating desktop applications, I will not discuss Libtool here. It's very simple to integrate into the rest of your build environment, however; if you're interested, visit <http://www.gnu.org/software/libtool/libtool.html>.

Figure 3-7 shows how Automake, Autoconf, and their respective input and output files interact. This may seem confusing right now, but this section will clarify things.

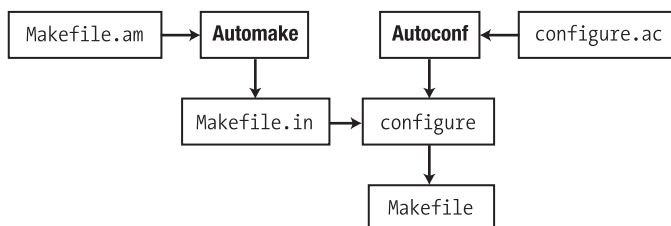


Figure 3-7. How GNU Autotools interact with each other

Autotools are very useful, but they're notoriously difficult to use properly for large-scale projects. This problem is escalated by the fact that different versions of the Autotools behave considerably differently. What worked fine on one version may fail miserably on another. The overview provided here remains simple enough to explain the tools' capabilities, and should work on any version of Autotools currently in use.

Automake

Automake takes a file called `Makefile.am` and turns it into a file called `Makefile.in`. This file will later be converted to `Makefile` when the configure script is written. `Makefile.am` is a very general way of describing how to build the project, and `Makefile.in` is essentially a makefile with some missing variables. When these missing variables are added by the configure script, it becomes the makefile for your project.

Automake can also determine which header files are included by each source file and can automatically maintain those as prerequisites as needed.

`Makefile.am` uses the same syntax as `Makefile`, and with a few exceptions, anything you include in `Makefile.am` will be added, verbatim, to `Makefile`. Because of this, you can add your own custom make targets to `Makefile.am` just as you would to `Makefile`, using the syntax discussed earlier. What makes `Makefile.am` different, then, are certain variables that have special meaning to Automake.

The first variable you'll want to set is `bin_PROGRAMS`. This is a list of all the executables to be created by your project. This will likely be only one, but you aren't limited to that. Gaim builds and installs two executables: `gaim` and `gaim-remote`, a small program that communicates with an existing Gaim session.

Variables in `Makefile.am` are defined just like any other variable:

```
bin_PROGRAMS = gaim gaim-remote
```

This tells Automake that it needs to build two programs: `gaim` and `gaim-remote`. `Makefile.am` then provides a brief description of how each of those programs is built using other variables.

The portion of a variable name in all capital letters is called a *primary*. These all have special significance to Automake. The lowercase words that prefix the primary further qualify it. `bin_PROGRAMS` implies a list of programs to be installed in the normal path for executables, typically `/usr/local/bin`. Programs normally executable only by the superuser (also called *root*) are typically installed into another directory, such as `/usr/local/sbin`. You could set `sbin_PROGRAMS` to list programs like this.

The descriptions you provide of how the programs in `bin_PROGRAMS` are built follow this same pattern. There are several primaries for each: most importantly, `SOURCES` and `HEADERS`. These are prefixed with the name of the program from `bin_PROGRAMS`:

```
gaim_SOURCES = file1.c \
               file2.c \
               file3.c
gaim_HEADERS = file2.h
```

This describes the program `gaim` as requiring three source files and one header file (the real Gaim is much larger). `gaim-remote` would be defined similarly, but the name of the program needs to be *normalized*. Because make disallows any characters other than alphanumeric and underscores in variables, you must convert any non-alphanumeric character to an underscore. For `gaim-remote`, this means the hyphen is converted to an underscore:

```
gaim_remote_SOURCES = gaim-remote.c \
                      prefix.c
gaim_remote_HEADERS= prefix.h
```

This is all that is required to have Automake create valid makefiles that build your application properly. Using the information determined by the `configure` script, the exact method of converting the provided sources and headers into binary executables will be determined. If your application requires other files, such as documentation, icon images, or sounds, Automake provides other primaries for those.

`DATA` is used for icons, sounds, and other data that needs to be installed without modification. `SCRIPTS` is similar. It is used for shell scripts. These are installed without modification also, but with different permissions. `MANS` is used for man pages, which are installed differently depending on the machine on which you compile the project. You can set the variable `EXTRA_DIST` with files that are distributed with your source, but not built or installed otherwise. This ordinarily includes your changelog and license agreement. Finally, for a project with subdirectories, set the `SUBDIRS` variable to contain them. Your top-level `Makefile.am` will probably be little more than setting `SUBDIRS`.

A final variable of note is `AM_CPPFLAGS`. This and the `LDADD` primary allow you to provide your own flags to the compiler and linker, respectively. This is especially useful when interacting with Autoconf. I'll discuss uses for these in the next section.

Anything else you put in `Makefile.am` will be copied exactly into the resulting `Makefile`. You can add your own targets directly to `Makefile.am`. Gaim does this with its `doc` target, allowing you to type `make doc` to generate documentation for Gaim's plug-in API.

Running automake from the top-level directory will read `Makefile.am` and process it and the `Makefile.am` files in each of its `SUBDIRS`. However, in order to do this, it first needs more configuration information as provided by `configure.ac`, the input to Autoconf.

Autoconf

Autoconf creates a `configure` script using the rules defined in `configure.ac`. The script it produces determines all the system-specific details of the environment the project is being built on. It determines what compiler to use, what options to pass to it, and where library dependencies are located. It also allows you to pass command-line arguments to `configure` that affect the way the project is built. The ultimate goal of `configure` is to use the `Makefile.in` files created by Automake and output `Makefiles`. However, the nature of `configure.ac` allows the `configure` script to do nearly anything.

Just as `Makefile.in` is essentially a makefile with a few fields that needed substitution, `configure.ac` is a shell script with a few fields that need replacement. However, where the fields replaced in `Makefile.in` are just variables, `configure.ac` uses macros. These macros—written in a macro language called `m4`—are very powerful and can be thousands of lines long. Autoconf and Automake provide a few standard macros, useful for building projects. You will rarely have to write your own macros (especially since you can write pure shell code in `configure.ac`).

The purpose of the `configure` script is to check various system properties. Autoconf makes checking for these properties simple; the hard part tends to be figuring out which properties to check for. Autoconf and Automake offer a large number of macros, only a fraction of which will be useful to you. You should be able to start small, with only a few mandatory macros, and then expand as your project becomes more complex.

The first required macro is `AC_INIT`, which initializes Autoconf. You provide to it your project name, the version number, and the e-mail address where users are expected to send bug reports. `m4` macros have a syntax very similar to C preprocessor macros, which in turn are

similar to C function calls. The main difference is that *everything* in an m4 macro is a string, so quotation marks aren't needed. However, arguments should always be quoted with square brackets such that Autoconf knows to treat it as a single argument. This is especially true when using other macros as arguments, as they may return code that contains commas. Quoting with brackets ensures that Autoconf knows that what's inside the quotes is all a single argument:

```
AC_INIT([gaim], [1.2.1], [gaim-devel@lists.sourceforge.net])
```

Next you must initialize Automake. `AC_INIT` has itself defined a few new macros you can use given the information provided to it. `AM_INIT_AUTOMAKE`, the macro that initializes Automake, also needs the package name and version number, so you can call the following:

```
AM_INIT_AUTOMAKE([AC_PACKAGE_NAME], [AC_PACKAGE_VERSION])
```

Next, you can have configure locate the command to use for the compiler. This is done with the `AC_PROG_CC` macro:

```
AC_PROG_CC
```

Finally, have configure create your makefiles. Unlike Automake, which uses one `Makefile.am` for each directory, there's only one configure script and likewise only one `configure.ac`. This script must create the makefile in every directory; this is done with the `AC_OUTPUT` macro:

```
AC_OUTPUT([Makefile
          src/Makefile
])
```

This command creates makefiles in the top-level directory and in `src/`. This very basic `configure.ac`, shown in its entirety in Listing 3-1, is enough to build a simple project with no dependencies. However, when you start depending on libraries, you'll need to check each of them and determine how to compile against them.

Listing 3-1. A Very Simple Makefile

```
AC_INIT([gaim], [1.2.1], [gaim-devel@lists.sourceforge.net])
AM_INIT_AUTOMAKE([AC_PACKAGE_NAME],[AC_PACKAGE_VERSION])
AC_PROG_CC
AC_OUTPUT([Makefile
          src/Makefile
])
```

Historically, every library has had its own way of properly locating itself and determining the correct flags to use when compiling or linking. The most common way was for each library to install an executable file somewhere within the user's `$PATH` to provide the version, compiler flags, and linker flags, depending on the arguments given. If I wanted to check if GTK+ was at least version 1.2, I would call

```
gtk-config --version
```

Then if I wanted to find out what compiler flags to use, I'd call

```
gtk-config --cflags
```


Obviously, having a separate executable for each library installed on your machine was cumbersome and resulted in a lot of duplicated efforts. Fortunately, many libraries (especially those related to GNOME, such as GTK+, GLib, and other related libraries) have moved to a common solution called `pkg-config`.

`pkg-config` does the same thing as `gtk-config`, but rather than require each library to provide its own binary, it has each library provide a `.pc` file that contains the relevant information and provides an identical interface for dealing with each library. Most importantly for this discussion of Autoconf, it also provides an Autoconf macro, called `PKG_CHECK_MODULES`:

```
PKG_CHECK_MODULES([GTK],[gtk+-2.0],[has_gtk=yes],[AC_MSG_ERROR([
* GTK+-2.0 is required to build Gaim])))
```

This macro call is a bit more complicated than the others seen. The purpose of this call is to determine what flags need to be sent to the compiler. The macro will return these in two variables. The first argument specifies a prefix to use for these variables; in this example, `GTK_CFLAGS` and `GTK_LIBS` will be set.

The second argument is a list of dependencies required from `pkg-config`. Here, it requires just the existence of one package. It can also be a list of packages (`[gtk+-2.0 glib-2.0]`) or a list of packages with required version dependencies (`[gtkspell-2.0 >= 2.0.2]`).

The third and fourth arguments are code to run when the package is found or not found, respectively. In this example, if GTK+ is found, it sets a variable called `has_gtk` to `yes`. If GTK+ is not found, another macro is executed. This macro is called `AC_MSG_ERROR`, and it prints an error message and exits the configure script. After your call to `PKG_CHECK_MODULES`, you'll want to be sure the information in `GTK_CFLAGS` and `GTK_LIBS` is included in the makefile.

Remember I said that `Makefile.in` is just a makefile with a bunch of variables missing. These variables are inserted to the eventual Makefile with the `AC_SUBST` macro:

```
AC_SUBST([GTK_CFLAGS])
AC_SUBST([GTK_LIBS])
```

These variables were set by `PKG_CHECK_MODULES`, and by calling `AC_SUBST`, they will be included in the resulting makefiles. However, you still need a way to provide these compiler options to the compiler; having them defined in a makefile doesn't do anything, otherwise.

Earlier I mentioned that `AM_CPPFLAGS` and the `LDADD` primary were important in interacting with Autoconf. I mentioned that those variables will be given as command-line options to the compiler and to the linker. In `Makefile.am`, you can set these to variables that will eventually be provided by the configure script:

```
AM_CPPFLAGS = $(GTK_CFLAGS)
gaim_LDADD = $(GTK_LIBS)
```

Doing this will pass the proper compiler flags along to the compiler. You can include a variable like this for each library you depend on.

Other Required Files

Automake requires that you add some additional files to your source tree other than `configure.ac` and `Makefile.am`. These files aren't actually used for anything but are included in the tarball distribution as standard documentation. These files are `NEWS`, `AUTHORS`, `ChangeLog`, and `README`.

These files have standard usage, but Automake does nothing to enforce that usage. Further, not following the recommended use of each file won't break the build, but it may confuse people accustomed to the standard. Gaim does not follow the standard. It uses ChangeLog as NEWS is supposed to be used, for instance.

ChangeLog is intended to log every change. Any time anyone changes a bit of code, it is supposed to be logged in ChangeLog. Further, the format of ChangeLog is specified. It should contain the date the change was made, the developer who made the change, the files that were affected, and the changes made in the following format:

```
Sean Egan <seanegan@gmail.com>
```

```
* gtkblist.c Fixed an off-by-one bug
```

However, because Gaim uses CVS for development, this sort of ChangeLog isn't really necessary (I'll discuss CVS later this chapter). Our ChangeLog serves the purpose that NEWS is supposed to: it provides a brief list of important, user-visible changes. Many of the changes in a new version of Gaim are never documented in NEWS or ChangeLog.

AUTHORS provides a list of the developers who worked on the project. This can be in any format you desire. README gives a brief description of what the project is and how to get started using it. Once you have created these four files, `configure.ac`, and a `Makefile.am` for every directory, you can bootstrap your build environment.

autogen.sh

Bootstrapping is the term for invoking the Autotools and creating a configure script and makefiles. The process is straightforward, but is usually delegated to a shell script to make the process even easier.

Create a file called `autogen.sh` in your top-level directory in an editor. Then add the following commands to it:

```
aclocal || exit;
autoconf || exit;
automake --add-missing --copy || exit;
./configure $@
```

Save the file and give it write permissions (`chmod +x autogen.sh`). The `|| exit` on each line tells the shell to exit the script if the command preceding it fails. If `autoconf` fails for some reason, `autogen.sh` will not attempt to run `automake`. The `$@` following `./configure` is a variable representing all the arguments provided to `autogen.sh`. This allows you to provide your configure command-line options to `autogen.sh`.

You know what `autoconf`, `automake`, and `configure` do, but you've not yet seen `aclocal`. Remember that `Autoconf` and `Automake` provide a ton of useful m4 macros for use in `Autoconf`? The `aclocal` command copies those macros into the project. This way, you can distribute your program to other people without them needing the same macros you have. The `--add-missing` and `--copy` arguments to `automake` are for similar reasons; they add automatically generated, required files to the project.

Using the Build Environment

Once you have your build environment set up, you should be able to run `autogen.sh`, which will bootstrap the environment, creating a `configure` file. When that's done it will execute `configure`. You should see `configure` performing its standard tests, which will probably look familiar to you if you've ever compiled an application before. Then, when `configure` is finished, you will have a `Makefile` in your current directory. Run

```
make
```

and the application will build. Then run

```
make install
```

as root to install it. You've created a working build environment and you now understand what makes it work and how to extend it as your project grows. Using Autotools makes your application very accessible to everyone. Autotools does the hard work of figuring out how to compile an application on a given machine.

Unfortunately, nobody will want to run your application if it's buggy. When bugs occur in your application, you will find the GNU Debugger, GDB, extremely useful in locating them.

GDB

The most notorious bug on UNIX systems is the *segmentation fault*, or *segfault*. A segfault occurs when your program attempts to access memory not allocated to it. The operating system terminates the application. Windows users know the same error as a general protection fault. Because C is very permissive in what it will allow you to do and because it makes heavy use of pointers, the segfault is one of the most common types of errors in C programming.

Fortunately, GDB makes it easy to track down segfaults. GDB has many other features as well, but I very rarely use it for anything other than tracking down segfaults. If you're interested in learning what else GDB can do, visit <http://www.gnu.org/software/gdb/gdb.html>.

First, before you can even start using GDB, make sure your application was compiled with debugging support. Recall that this is done by setting `-g` to the linker. If you're using Autotools to create your build environment, set `-g` in `AM_CPPFLAGS`. This will provide the appropriate debugging hooks within the resulting executables.

Running Your Application Within GDB

There are two ways of capturing information about a crash in GDB. The first way is to run the application within GDB. When the operating system attempts to kill the segfaulting application, GDB will notice and provide an interface to debug it. To run an application within GDB, say `Gaim` for example, run

```
gdb gaim
```

GDB will launch and display a copyright notice and a prompt:

```
GNU gdb 6.3-debian
```

```
Copyright 2004 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are
```

```
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library
"/lib/tls/i686/cmov/libthread_db.so.1 ".
```

```
(gdb)
```

(gdb) is the GDB prompt. To run Gaim, type `run` at the prompt. Running Gaim in GDB is useful for reproducible errors; you can launch it and quickly do exactly what is required to make it crash. When it does crash, you will be returned to the (gdb) prompt.

Analyzing Core Dumps with GDB

For non-reproducible errors, though, it's inconvenient to have the debugger constantly running. When the application is killed it will leave a *core dump* behind. A core dump provides essentially a snapshot of the process's memory when it crashed. Because these files are potentially very large, many UNIX distributions disable them by default. However, if your application is crashing sporadically with no obvious cause, it may be worthwhile to enable core dumps. You can do this with the `ulimit` command:

```
ulimit -c unlimited
```

When you enable core dumps and your application crashes, it will leave behind a core dump in the directory from which the application was launched. The name of the core dump varies from system to system, but is almost always a combination of the word “core,” the name of the application that produced it, and the process identifier (PID) of the particular process that caused it. When you have a core dump, you can load it into GDB by associating it with a given application on the command line:

```
gdb gaim core.13905
```

This tells GDB to load the provided core dump that was produced by the Gaim command. GDB will show the same copyright notice shown in the previous section and bring you to a (gdb) prompt. Whether you've run Gaim and had it crash or you've loaded the core dump of an already crashed Gaim, you now have access to the same debugging features.

Debugging Segfaults

The most useful of the debugging features is the backtrace. A backtrace shows the function that Gaim crashed in, and a list of all the function calls that led to that function, starting at `main()`. For example, if `main()` calls `functionA()` and `functionA()` calls `functionB()`, the backtrace will show this. You can get a backtrace from the (gdb) prompt with the `bt` command. In the next chapter, I will introduce you to Gaim development with a simple example plug-in. For demonstration here, I've intentionally introduced a bug to it. Watch how I isolate the bug with GDB. First, I ask for the backtrace:

```
(gdb) bt
#0  0x40559a4e in g_strdup () from /usr/lib/libglib-2.0.so.0
#1  0x4030f0b1 in gtk_window_set_title () from /usr/lib/libgtk-x11-2.0.so.0
```

```
#2 0x401b0fc1 in gtk_dialog_new () from /usr/lib/libgtk-x11-2.0.so.0
#3 0x401b1053 in gtk_dialog_new_with_buttons () from /usr/lib/libgtk-x11-2.0.so.0
#4 0x080dd199 in gaim_gtk_notify_message (type=GAIM_NOTIFY_MSG_INFO,
    title=0x1 <Address 0x1 out of bounds>, primary=0x0, secondary=0x40d3c900 "I've
    written a plugin", cb=0, user_data=0x0) at gtknotify.c:113
#5 0x0807f3b2 in gaim_notify_message (handle=0x0, type=GAIM_NOTIFY_MSG_INFO,
    title=0x1 <Address 0x1 out of bounds>,
    primary=0x40d3c915 "Hello World!", secondary=0x0, cb=0, user_data=0x0)
    at notify.c:55
#6 0x40d3c842 in plugin_load (plugin=0x81d9418) at helloworld.c:9
#7 0x0807fed6 in gaim_plugin_load (plugin=0x81d9418) at plugin.c:340
#8 0x080e583e in plugin_load (cell=0x85f4928, pth=0x0, data=0x85f4268) at
    gtkprefs.c:1960
```

I've shown only the first eight lines of the backtrace here. The actual backtrace is 34 lines long, going all the way back to `main()`. Here we see that the segfault occurred in `g_strdup()`. `g_strdup()` was called by `gtk_window_set_title()`, which was called by `gtk_dialog_new()` and so on.

Tip Sometimes it's possible to corrupt memory at one point in your code but not cause a crash until much later, typically in code entirely unrelated to the bug. This makes debugging far more difficult. Because of this, on Linux systems, you can specifically tune memory allocation functions to detect errors sooner with environment variables. Set `MALLOC_CHECK_` to 2, and your program will segfault as soon as a memory error is detected. Running this way normally is not recommended, however, as it is considerably slower.

Although it's not unheard of for GTK+ itself to have a bug, it's usually quite stable. Most of the time when you see GTK+ calls (or calls to other libraries) at the top of your backtrace, it's the result of providing the library with bad data further down. Garbage in, garbage out, as they say. Here, line 4 looks particularly suspicious the way it reads `title=0x1 <Address 0x1 out of bounds>`. Also, I recognize it as a function within Gaim, which is generally less stable than GTK+. I decide to look at line 4 more carefully with the `frame` command:

```
(gdb) frame 4
#4 0x080dd199 in gaim_gtk_notify_message (type=GAIM_NOTIFY_MESSAGE_WARNING,
    title=0x1 <Address 0x1 out of bounds>, primary=0x0, secondary=0x40d3c900 "I've
    written a plugin", cb=0, user_data=0x0) at gtknotify.c:113
113  dialog = gtk_dialog_new_with_buttons(title ? title : GAIM_ALERT_TITLE,
```

The `frame` command shows the function that was called, `gaim_gtk_notify_message()`, and the arguments provided to it. It shows the exact relevant line in the source and even prints it to the screen. You can see here that the variable that came in (`title=0x1`) was passed on directly to `gtk_dialog_new_with_buttons()`. Again, Garbage In, Garbage Out. Because the `0x1` argument was passed into this function, I check the function that called it:

```
(gdb) frame 5
#5 0x0807f3b2 in gaim_notify_message (handle=0x0, type=GAIM_NOTIFY_MSG_INFO,
    title=0x1 <Address 0x1 out of bounds>,
    primary=0x40d3c915 "Hello World!", secondary=0x0, cb=0, user_data=0x0)
    at notify.c:55
55     info->ui_handle = ops->notify_message(type, title, primary,
```

Again, the bogus variable was fed as input to this function. In some instances, you may be able to design your functions to be more tolerant to invalid inputs. For example, it's very common to check for NULL, which is never a valid memory address. However, any variable may be invalid sometimes, and valid other times. It changes on a process-to-process basis, depending on what memory is allocated by the operating system. Because `gaim_notify_message()` is also given the invalid variable, I try on layer higher:

```
(gdb) frame 6
#6 0x40d3c842 in plugin_load (plugin=0x81d9418) at helloworld.c:9
9         gaim_notify_info(NULL, title, "Hello World!", "I've written a plugin");
```

The 0x1 variable seems to be sourced here. There is no title input, but it outputs a local variable called `title`. Plus, `helloworld.c` is my newly written plug-in that hasn't been tested yet. This seems like a likely place for the bug to be. I check out the value of the `title` variable with the `print` command.

```
(gdb) print title
$2 = 0x1 <Address 0x1 out of bounds>
```

`print` will print any variable that's valid in the current scope, that being the function specified by the frame command. Here it shows me that the value of `title` is indeed 0x1. I'll need to check the source code to see where that value comes from. I open up `helloworld.c` and navigate to around line 9. Reading the function, I see the following:

```
char *title = 1;
```

That's where the 0x1 has come from. I change the value of `title` to 0, which is a valid argument meaning "no title." I recompile the plug-in, and it loads without error.

One of the most cited advantages of the open source development process is that it allows an extraordinary number of people the ability to locate and fix bugs using tools like GDB. However, the task of getting such a large number of people to easily work simultaneously on the same code is not trivial. Fortunately, there are systems in place that attempt to make the task as painless as possible. The system Gaim uses is called CVS.

CVS

The Internet enables open source development to succeed perhaps more than any other tool. Whereas before the Internet, developers who wanted to work on the same code would need to be at the same computer, the Internet allows developers from all over the world to discover and contribute to any project run by any stranger anywhere. Over 200 people have contributed to Gaim this way. CVS is the system that enables hundreds of developers access to the same source code and manages multiple people working on it simultaneously over the Internet.

Versioning

However, the primary goal of CVS is versioning. Think of versioning as a superpowerful “undo” feature. Anytime you make a change to the code, that change gets recorded in CVS as a new revision, and CVS allows easy ways to revert the code to its state at any time prior.

Imagine you start noticing a new bug in your program, and you can’t track it down with GDB alone. However, you know it’s just sprung up in the past three days. You can tell CVS to isolate for you all the changes made in the past three days. When you read just those changes, you easily find the bug, fix it, and commit it back to CVS so that others can use it.

Versioning also includes *tags*, which let you mark particular moments in the code with some label. For example, every time Gaim makes a release, after we’ve all made final changes to the code in CVS, we tag it with the version number of the release. This way, we have a very easy way of telling CVS to give us the exact state of the code included in the 0.60 release, for instance. A related feature to tags is *branches*.

Branches

Sometimes you need to branch off of your development sources and create a separate parallel branch of your development tree. For instance, as I write this, Gaim development is happening in two branches simultaneously. One branch is used for stable, bug-fix releases, while active development on what will become Gaim 2.0.0 is happening in the main branch, called HEAD. This is the most common use of branches; you can create stable releases off of a stable branch, while doing major development on HEAD. Branches can also be used to test experimental code without affecting the rest of the code at large.

CVS is a very useful tool for allowing collaborative development. In the next few sections, I will discuss how you, as a developer, will use CVS to access a single source repository with fellow developers.

Using CVS

When you need to work on a project in CVS, you check it out of the central repository. You then have a local copy of the source on your machine. When you’re satisfied with your changes, you commit them back to the central repository. Each developer maintains his own source trees, and CVS intelligently merges them together when needed. In this section, I’ll walk through the typical usage for most developers using CVS, explaining how to use its commands and what they’re good for.

Logging In

The first thing to do is tell CVS what server to use and where on the server your repository is located. You can do this by setting the CVSROOT environment variable, or by setting it on the command line. I prefer the second option and specify the CVSROOT on the command line.

Many projects, including Gaim, have anonymous CVS. The Gaim developers strongly discourage using builds from CVS in favor of the most recent release. However, people who wish to start contributing to development prefer to retrieve the source from CVS to make sure they’re not duplicating effort or working on old code. To facilitate this, many projects will allow everyone to read from their CVS repository, but not write to it. If you’re using anonymous CVS, you will be provided a specific CVSROOT username and password, which may be different from

what the developers use. You should consult whoever maintains your CVS server about how to access it. To log in to Gaim's anonymous CVS, run

```
cvs -d':pserver:anonymous@cvs.sourceforge.net:/cvsroot/gaim' login
```

It will ask you your password. SourceForge.net's anonymous CVS does not have a password; just hit Enter. Once you're logged in, you can download the latest source code from the central CVS repository.

Checking Out

In your CVS repository, you can maintain several modules. These modules are normally entirely unrelated to each other; and so they're kept separate so that developers working on one module are not affected by developers working on another module. Gaim's CVS repository hosts two modules: `gaim`, where the Gaim source code is stored, and `web`, where the code for our website is stored.

You must again specify the correct `CVSROOT` either in the environment variable or the command line. Then, rather than `login`, the command is `checkout <modulename>`. To check out Gaim from anonymous CVS, run

```
cvs -d':pserver:anonymous@cvs.sourceforge.net:/cvsroot/gaim' checkout gaim
```

Common commands have abbreviations, so instead of `checkout`, I could use `co`:

```
cvs -d':pserver:anonymous@cvs.sourceforge.net:/cvsroot/gaim' co gaim
```

This will create a directory in your current folder with the name of the module, in this case `gaim`. This directory will store all the files in the repository and files used internally by CVS. Because it stores its own information, CVS will no longer require you to specify a `CVSROOT` when working within this directory; it saves that information itself.

After CVS creates the directory, it will begin to populate it with the files in the repository. You will see them listed, one at a time, each prepended with `U`. This `U` is used to describe what is happening to the file, and is mostly useful while updating your local tree.

Updating

Gaim has consistently been the most actively developed open source project out of nearly 100,000. With development that active, you can imagine that it won't take long for your freshly checked-out tree to become out-of-date. Fortunately, CVS can update your tree to be up-to-date with the current tree. This has two significant advantages.

Advantages of Updating

First, CVS will send you only the parts of the file that have changed. Gaim's source distributions are about 8 megabytes; the entire source tree is larger. It would be a waste to receive all 8MB. By sending only what's changed, you save time and bandwidth.

The second advantage is made possible by the first. Because CVS can keep track of the changes you've made to your local tree and is provided the changes that have been made since your last update, CVS can intelligently merge your changes along with the changes of other developers. Suppose I need to work on `gtkaccount.c` for a while; I would update it from CVS

and make my changes. While I was working on my changes, another developer committed a few bug fixes to that file.

For instance, assume I'm working on some changes to the `gaim_conv_write()` function in `gtkconv.c`. This is a complicated change, and it takes a few days to complete. In the meantime, another developer makes a one-line bug fix to another function in `gtkconv.c`. The version of the source code I'm working on predates the fix.

Without CVS, I would have to examine the file, see the changes he made, and manually incorporate them into my tree before uploading the file. CVS can figure out when two changes are made to different parts of the file and automatically merge them in your local tree. It will warn you if there are any errors in merging, and when you're happy that there are none, you can commit your changes.

How to Update

To update your local tree to the current repository, run

```
cvs update -d
```

or, shorter,

```
cvs up -d
```

The `-d` option tells CVS to create new files in the local tree if they've been added to the repository. It's usually unnecessary, but without it you often won't discover new files have been added until make fails. Sometimes you want to completely revert to current CVS, and remove all your local changes. I usually do this when I realize I've taken the completely wrong approach to something and I want to start over. A few commands used together will result in you entirely restoring your local tree. Fortunately, these commands are very easy to remember:

```
cvs up -dCRAP
```

Updating to Past Revisions

Lastly, I've explained that CVS's primary purpose is to provide versioning. CVS makes it easy to restore your source code from some time in the past. If you're interested only in one file and you know its revision history, you can request a version of it by its revision:

```
cvs up -r 1.175 gtkaccount.c
```

This reverts `gtkaccount.c` to revision 1.175. I most typically look at revision information from a Web application called ViewCVS. I'll discuss it briefly later. More useful than updating to a specific revision is updating to a specific tag. Revision numbers and tags are treated the same way, so to update my entire tree to when it was tagged `v1_1_1`, I'd run

```
cvs up -r v1_1_1
```

Branches are considered just special types of tags, so if you need to sync your tree with a different branch, use the branch name as a tag name. It's also sometimes useful to revert your tree to its state at some arbitrary moment in the past. This uses the `-D` argument, which is very flexible in how it lets you specify dates. All the following are legal CVS commands:

```

cvs up -D "April 5, 2003 12:00:00"
cvs up -D "yesterday at midnight"
cvs up -D "5 weeks ago"
cvs up -D "last week"
cvs up -D "a fortnight ago"

```

You can essentially describe in plain English to CVS when you want a tree from, and it will comply.

Output from `cvs update`

After you issue your command to update CVS, you'll see a list of the files it's updating. Like the output from `checkout`, each file will be prepended by a single letter representing its status. Table 3-8 shows what these characters mean:

Table 3-8. *Output from `cvs update`*

Character	Meaning
U	The file was brought up-to-date with no problems. This is what you see when checking out.
P	This is effectively the same as U, but where U indicates the entire new version of a file has been transferred, P indicates that only a patch containing the relevant changes was transferred.
A	This file isn't in CVS yet, but you've told CVS to add it. It will be added the next time you commit changes.
R	This file is in CVS, but you've told CVS to remove it. Again, it will be removed from CVS next time you commit. A and R serve as reminders to commit.
M	You've made modifications to this file since your last update. CVS was able to merge your changes with any made in CVS.
C	You've made modifications to this file, but CVS was unable to reconcile your changes with changes made by other developers in CVS. (See the "Dealing with Conflicts" section that follows this table.)
?	This file is in your local tree, but CVS knows nothing about it. The file <code>.cvsignore</code> contains all the files whose presence CVS should ignore in a directory. This keeps the number of files marked with ? to a minimum.

Dealing with Conflicts

If you see a C in your output, it means there was a conflict in merging your changes with others. It doesn't happen often; it typically happens only when two people make changes to the same exact lines of code. You will not be able to commit your code to CVS until you resolve the conflict. CVS makes this easy.

CVS will mark up your code to show exactly where the conflict occurred. Lines will be added at a location that shows what code exists at the location in current CVS, and what you think should be there. In the example here, another developer and I decided to debug the same bit of code, but went about it slightly differently:

```
<<<<<< gtkaccount.c
    gaim_debug_warning("account", "This doesn't look right.");
=====
    gaim_debug_warning("account", "account_name is %s\n", account_name);
>>>>>> 1.154
```

I will explain the Gaim API in the next chapter. After reading that, you will understand what `gaim_debug_warning()` means; for now, know that it's a function call. This line caused a conflict because while we both added a call to `gaim_debug_warning()` since my last CVS update, we used a different message. My code is listed first, between the `<<<<<<` and the `=====`. The code currently in CVS is listed second. CVS can't figure out which line is correct. I figure that the second code is more useful for debugging, I delete everything but that code, and can safely commit.

If you have made changes to your checked-out code and then updated to make sure your changes are against the most up-to-date version of the code, and you want to release your changes back to the public, you have two ways of doing it. If you have commit access to CVS, you can commit the changes yourself. If you do not, you will have to create a patch and send it to someone with commit access for consideration.

Patches

Patches are descriptions of what text has changed in a file. They are the primary way that contributors to Gaim provide their changes upstream to the core Gaim developers. Patches are plain text files that are easy for humans to read and are easily created and processed with the `diff` and `patch` tools, respectively.

Anatomy of a Patch

Gaim developers (and most other developers) prefer their patches to be in unified diff format. This format is very straightforward to read. When CVS creates a patch, it shows the files affected, what revision of the file the patch was made from and when it was made, the line numbers where the changes occur, and the changes themselves. Listing 3-2 is part of a patch submitted by Don Seiler.

Listing 3-2. One “Hunk” of a Patch

```
Index: src/accountopt.h
=====
RCS file: /cvsroot/gaim/gaim/src/accountopt.h,v
retrieving revision 1.9
diff -u -p -r1.9 accountopt.h
--- src/accountopt.h      6 Mar 2005 06:14:26 -0000    1.9
+++ src/accountopt.h      14 Mar 2005 22:22:41 -0000
@@ -133,8 +133,8 @@ GaimAccountOption *gaim_account_option_s
 * The list passed will be owned by the account option, and the
 * strings inside will be freed automatically.
 *
- * The list is in key, value pairs. The key is the ID stored and used
- * internally, and the value is the label displayed.
+ * The list is a list of GaimKeyValuePair items. The key is the ID stored and
```

```
+ * used internally, and the value is the label displayed.
*
* @param text      The text of the option.
* @param pref_name The account preference name for the option.
```

Each change in a patch file is called a *hunk*. There can be several hunks from the same file. The first line of Listing 3-2 shows the file affected. Underneath the line of equals signs can be any arbitrary text at all. If you wanted, you could edit text located here to provide a description of what your changes do. CVS automatically includes some useful information, such as when the patch was made, and off of what revision of the file.

This line is the beginning of the first hunk, as signified by the double @ sign:

```
@@ -133,8 +133,8 @@ GaimAccountOption *gaim_account_option_s
```

In this example, 133,8 describes the location of the original code; 133 is the line number where the original code started, and 8 is the number of lines in the original code. The second set of numbers is the same thing, but for the new code.

■ **Note** Gaim developer (and this book’s technical reviewer) Nathan Walp is also called a “hunk.”

In this example, the two sets of numbers are equal; two lines were deleted, and two lines added. However, if I add 10 lines of code in my patch, where formerly there were only two, the two lengths would be different. Similarly, if previous hunks changed the number of lines before this, the two line numbers would be different.

The text after the second @@ can be anything; here it lists the function this hunk is within to make it easier to locate the appropriate text in an editor. This is an option provided by a command-line option to `diff`.

After the hunk header, the patch shows the changes made. It will show a few lines on either side of the actual change as a courtesy, to provide context. Lines that are removed are prefixed with -, lines that are added are prefixed with +. When lines are merely changes, as they are here, they’re represented as lines being removed, followed by lines being added.

Following this would be a second hunk for that file, if it were changed in more than one place, or another `Index:` line marking the beginning of changes to a new file. If this were the only change made, Listing 3-2 would be the entire contents of the file.

Creating a Patch

Creating a patch from CVS is easy. Make the changes to your local tree, make sure it’s synced to current CVS with `cvps up`, and run

```
cvps diff -pu
```

This uses two command-line options: `-u` requests a unified patch in unified format (exemplified in Listing 3-2), and `-p` asks `diff` to include the function name in the hunk header, as seen in the example.

This command will print the patch to your console. In order to send this to a developer, you’ll likely want to save it to a file. This can be done with the UNIX shell’s redirection functionality:

```
cvs diff -pu > gaim_account_option_list_ui.diff
```

A good filename for a patch includes the project the patch is for, a brief summary of what it changes, and a `.diff` extension.

You can also create a patch between any two tags or dates in the repository's history using the same context as updating to some other point in the repository's history. The main difference is that you can specify only one tag or date to see changes between that point and your local tree, or two tags or dates to see the changes between any other two points. For instance, to generate a patch of all the changes between Gaim v1.0.0 and Gaim v1.0.1, run the following:

```
cvs diff -pu -r v1_0_0 -r v1_0_1 > gaim-1.0.1-incremental_patch.diff
```

Patches like this are often useful for people who download releases, so that they needn't download a new 8MB tarball, and can instead download only new changes.

Applying a Patch

If you are a core developer with commit access to CVS, you'll probably be responsible for reviewing, applying, and committing patches that other people submit. Reviewing can be done by reading the patch file itself. The unified diff format makes it easy to see exactly what changes. Applying is also easily done with the patch utility.

`patch` takes a patch file from standard input, so it is usually fed via the same redirection method used to redirect the output from `cvs diff` to a file. `patch` also requires that you tell it where the files being patched are located relative to the current location. This is done with the `-p` option. However, CVS patches are almost always both generated and applied from the top-level directory of the project, so you don't have to understand exactly what `-p` is used for. You can always use `-p0`:

```
patch -p0 < gaim_account_option_list_ui.diff
```

This will affect the changes from `gaim_account_option_list_ui.diff` to your local tree. `patch` is very intelligent; it will be able to figure out what to change even if your local tree has changed considerably since the patch author submitted the patch. However, if your tree has changed too much, `patch` may skip a chunk and save the rejected hunk to a `.rej` file it will output on the console.

Tip If you aren't sure if a patch will apply cleanly, try using the `--dry-run` command-line option (`patch --dry-run -p0 < gaim_account_option_list_ui.diff`). This will make sure each hunk applies before modifying any file.

After you commit the patch to your local tree, or after you have made your own changes and you want to share the changes with everyone else, you need to commit your changes to the CVS repository.

Committing

When you commit changes to your CVS repository, you make them publicly available as part of the official source code. From CVS, the code will be in subsequent releases, and other developers can work with it. Committing code to CVS is simple:

```
cvcs commit
```

or the abbreviated form:

```
cvcs ci
```

CVS will determine all the changes that need to be made, and open an editor asking you to enter a log message. CVS has its own changelog; when someone makes a commit, they write a log message about it. To be useful, this commit usually states what the patch does, who wrote it, and what potential problems it has.

Adding and Removing Files

You will need to add and remove files from the CVS repository at times. These tasks are done with the `add` and `remove` commands. To add a file to CVS, put it in your local tree and run

```
cvcs add newfile.c
```

If you need to remove a file, delete it first, then run

```
cvcs remove oldfile.c
```

Neither of these commands will actually take effect until you run `cvcs commit`, providing you an opportunity to change your mind. Also, files that are removed from the repository retain all their versioning information. The file is simply moved from the main directory to the `Attic`, a location reserved for removed files.

Tagging and Branching

When you need to make a release, or whenever the code is at some point that you want to be able to access easily, you create a tag. Creating a tag is very straightforward. To tag for Gaim release version 1.2.1, I would call

```
cvcs tag v1_2_1
```

You can also retroactively create tags at various other points in the past, using the already familiar `-r` and `-D` options:

```
cvs tag -D "November 18 at 11:18pm" OLDER_TAG
```

You can delete old tags; this is in case right after tagging for release you realize you forgot an important change. You can delete the tag, commit the change, and then retag with the same tag:

```
cvs tag -d v1_2_3
```

Branches, discussed briefly earlier, are conceptually nothing more than a special type of tag. A branch is sometimes called a *sticky* tag because when you update to a sticky tag, the tag remains with your local tree until you run `cvs up` with the `-A` argument (which is included in the `cvs -dCRAP` command discussed earlier). This means that when you make commits, they all affect the sticky tag.

To create a branch, call `cvs tag` with the `-b` argument:

```
cvs tag -b experimental_branch
```

This creates a new branch, called `experimental_branch`. When you do `cvs up -dr experimental_branch`, you get the code for that branch, and any code you commit goes to `experimental_branch`. There's no way to automatically merge changes between two branches. You must make changes to each branch individually.

Many of the commands I've demonstrated use revision numbers. It's possible to generate and view file history with the `cvs` command itself, but it's somewhat awkward and difficult to master. This job is made far easier with a Web application called ViewCVS.

ViewCVS

ViewCVS is a Web application installed by whomever administers your CVS server. Gaim uses ViewCVS as provided by SourceForge.net (discussed in the next section). ViewCVS provides a really simple way to view your project's versioning history. Gaim's ViewCVS is located at <http://cvs.sourceforge.net/viewcvs.py/gaim/>, shown in Figure 3-8.

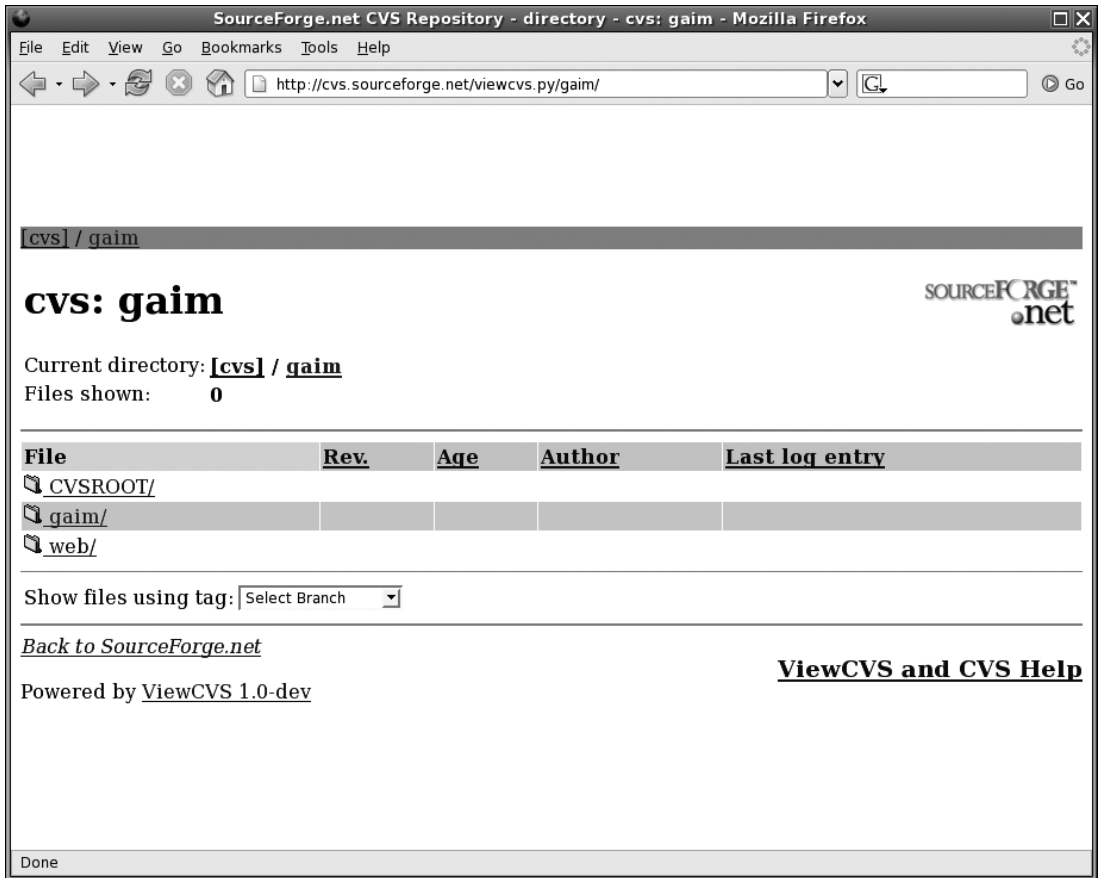
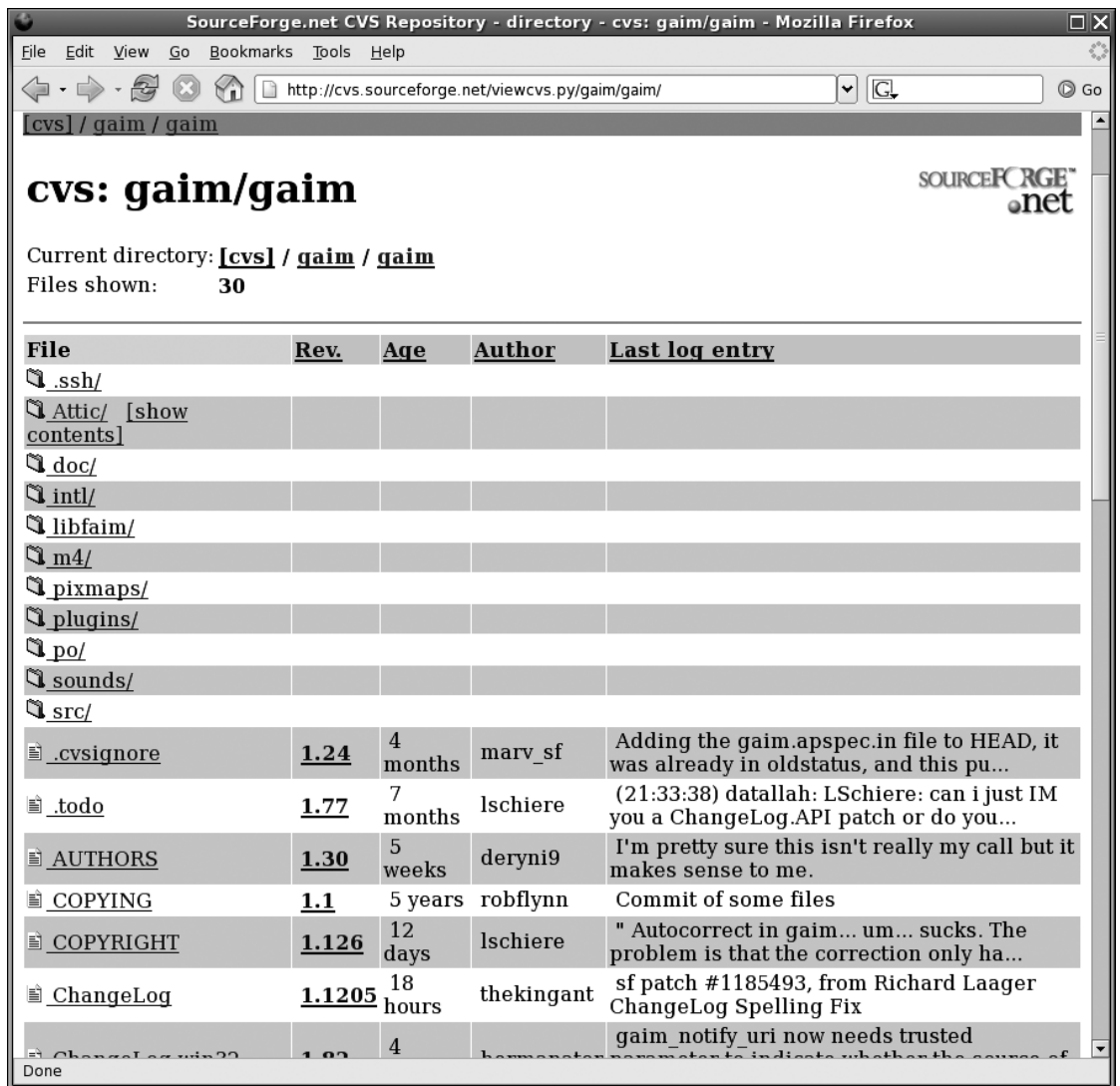


Figure 3-8. The top-level view of Gaim's CVS repository shows the two modules contained within it.

The top-level view in ViewCVS shows the two modules in Gaim's CVS repository. The gaim module is more interesting. Click its link to view the module, shown in Figure 3-9.



SourceForge.net CVS Repository - directory - cvs: gaim/gaim - Mozilla Firefox

http://cvs.sourceforge.net/viewcvs.py/gaim/gaim/

cvs: gaim/gaim

Current directory: **[cvs] / gaim / gaim**

Files shown: **30**

File	Rev.	Age	Author	Last log entry
.ssh/				
Attic/ [show contents]				
doc/				
intl/				
libfaim/				
m4/				
pixmap/				
plugins/				
po/				
sounds/				
src/				
.cvsignore	1.24	4 months	marv_sf	Adding the gaim.apspec.in file to HEAD, it was already in oldstatus, and this pu...
.todo	1.77	7 months	lschiere	(21:33:38) datallah: LSchiere: can i just IM you a ChangeLog.API patch or do you...
AUTHORS	1.30	5 weeks	deryni9	I'm pretty sure this isn't really my call but it makes sense to me.
COPYING	1.1	5 years	robflynn	Commit of some files
COPYRIGHT	1.126	12 days	lschiere	" Autocorrect in gaim... um... sucks. The problem is that the correction only ha...
ChangeLog	1.1205	18 hours	thekingant	sf patch #1185493, from Richard Laager ChangeLog Spelling Fix
ChangeLog.win32	1.02	4	harmenstap	gaim_notify_uri now needs trusted parameters to indicate whether the source of

Done

Figure 3-9. The *gaim* module

This view shows all the files and directories in this, the top-level directory of the module. Notice it also shows Attic, where old, removed files retain their versioning information.

Each file lists its current revision, how long ago that revision was made, who made it, and the changelog entry for it. Let's take a closer look at `Makefile.am`, a file I explained how to create earlier in this chapter. Clicking on the revision number shows the syntax-highlighted content of that revision, as seen in Figure 3-10. Clicking the filename will show the full history of the file, shown in Figure 3-11.



Figure 3-10. The contents of a file in CVS

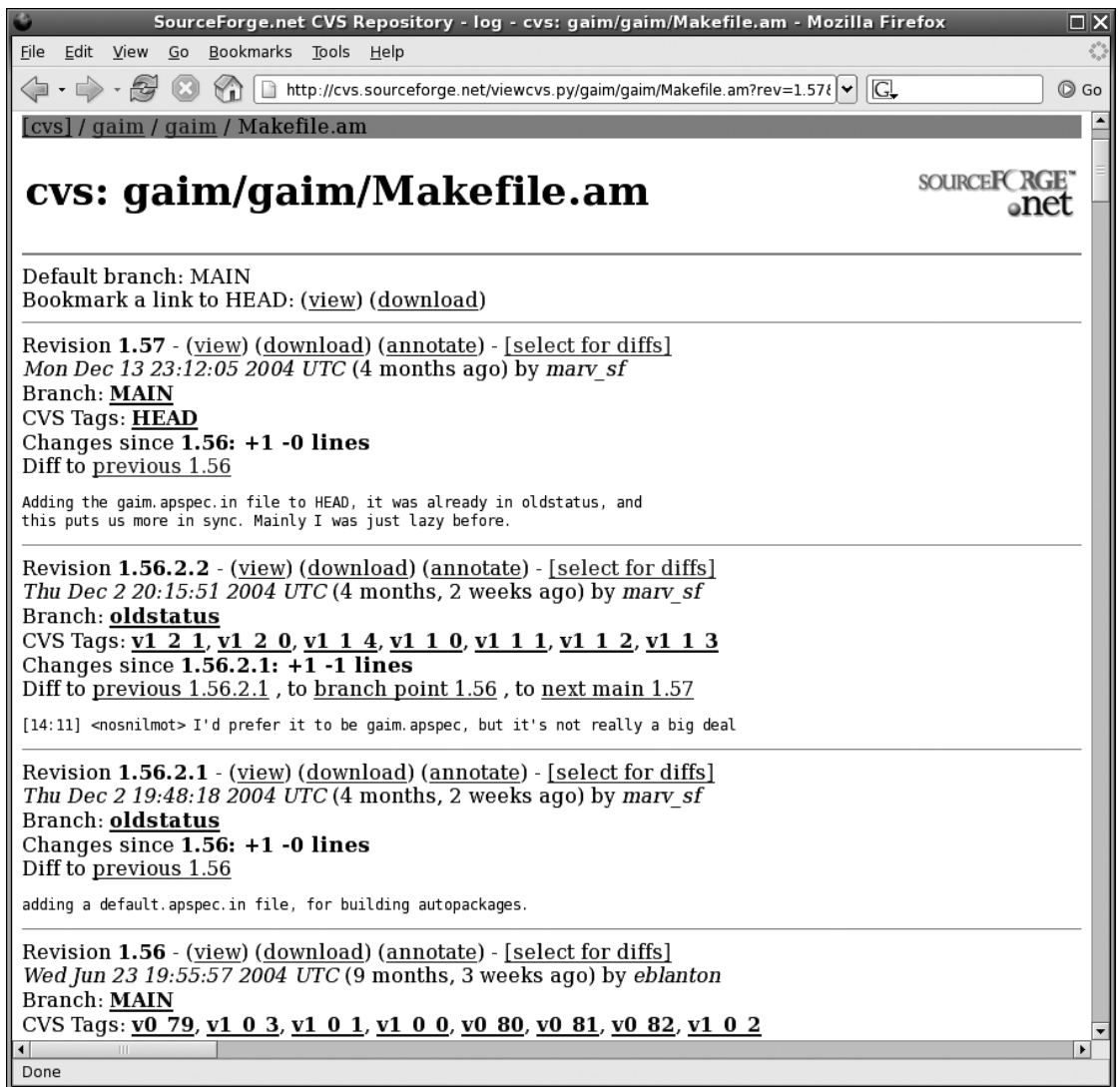


Figure 3-11. The full history of *Makefile.am*

In Figure 3-11, you see a list of each commit that affected *Makefile.am*. Each entry in the list contains the revision number, the date and time it was committed, who committed it, what branch the commit was to, what tags apply to it, and a link to show a patch to the previous revision. Patches in ViewCVS are slightly easier to read than they are in a typical text editor; they're color-coded to show you exactly what's been removed, added, and modified. The patch between revision 1.56 and 1.57 of *Makefile.am* is shown in Figure 3-12.

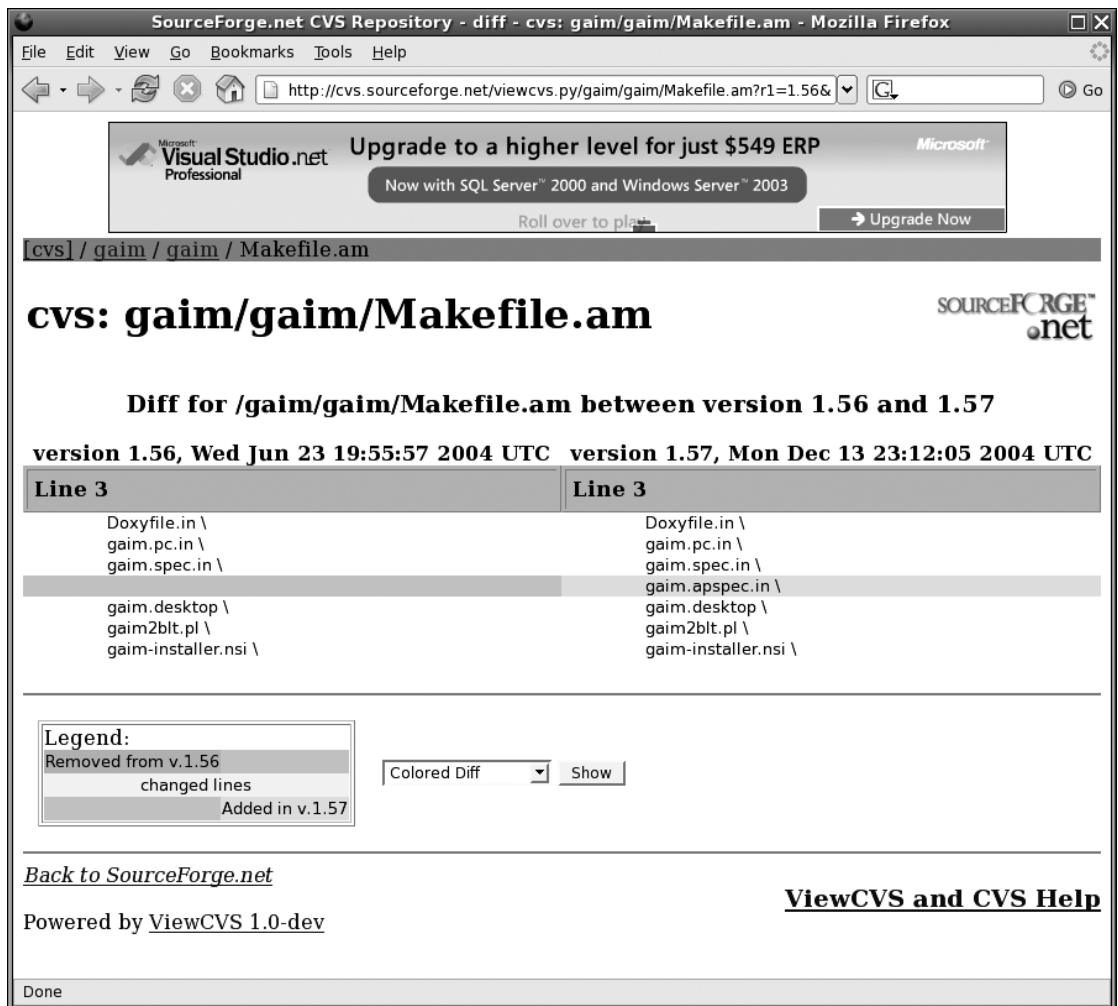


Figure 3-12. A patch between revision 1.56 and 1.57 of *Makefile.am* as seen in ViewCVS

ViewCVS provides a convenient interface to manage the versioning control of your project. In fact, many of the most useful tools used to manage your open source project are available on the Web. SourceForge.net is a free service that provides ViewCVS and a number of other useful means of managing your open source project.

SourceForge.net

Gaim is hosted by SourceForge.net, a site dedicated to providing open source projects with a number of free tools for managing those projects. The tools include CVS (including ViewCVS), which I've just discussed, mailing lists, project Web space, and a file-release system with a large number of mirrors around the world. The most notable unique feature to SourceForge.net is its tracker system.

Note The concept of a tracker is not unique to SourceForge.net, but SourceForge's implementation of a tracker is. Perhaps the most popular open source tracker software is Bugzilla, from the Mozilla project, which you can read about at <http://www.bugzilla.org>.

A *tracker* is used to track issues, most prominently bug reports that lead to the alternate name “bug tracker,” but anything can be considered an issue. Gaim has trackers for bugs, feature requests, submitted patches, third-party plug-ins, translations, and other items.

The best way of communicating to Gaim is to submit an item to one of our trackers. If you have found a bug, file a bug report to the bug tracker. If you wish to request a feature, file a request in our RFE (Request for Enhancement) tracker. Tracker items will remain open until they've been resolved, meaning it won't be forgotten as often happens to items submitted via e-mail or some other mechanism.

In this section, I will describe the process of working with the SourceForge.net trackers, from both the perspective of the user submitting an item, and of a developer resolving it.

Submitting a Tracker Item

A Gaim user will find a Bug Reports link in the menu along the right side of the Gaim Web page at <http://gaim.sourceforge.net>. This leads directly to the Submit New Issue page for the bug tracker, shown in Figure 3-13. Alternatively, a user can reach the page from the SourceForge.net project page (<http://sourceforge.net/projects/gaim>), which is also linked to from the Gaim Web page.

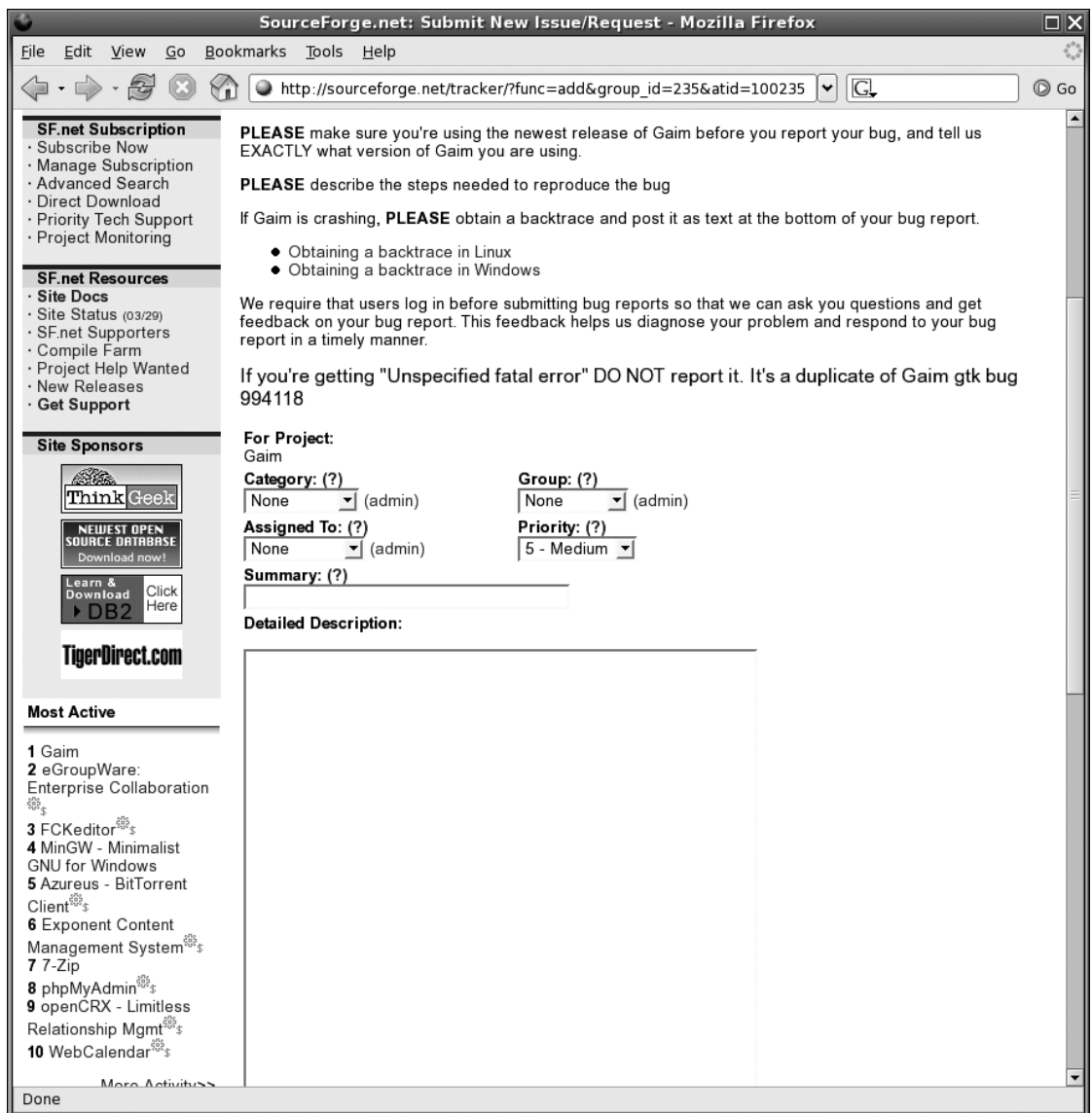


Figure 3-13. SourceForge.net's Submit New Issue page

The SourceForge.net project page, shown in Figure 3-14, has a menu along the top. This menu links to the various trackers and other services provided by SourceForge.net. Clicking Bugs brings you to the bug tracker, from which you can click Submit New, to get the same Submit New Issue page shown in Figure 3-13.

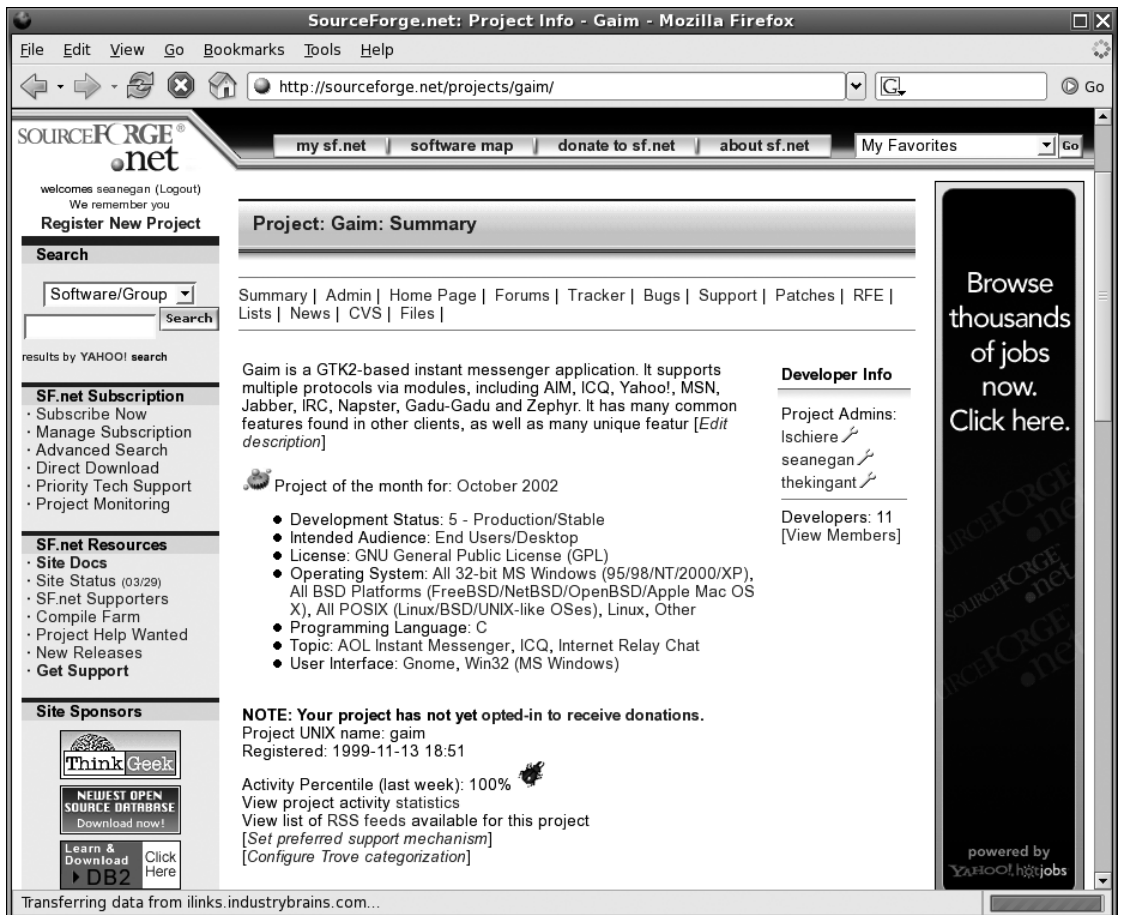


Figure 3-14. Gaim's SourceForge.net project page

The page shown in Figure 3-13 is specifically for the bug tracker; however, the same page is used for any tracker. Submitting a patch or a feature request uses this same interface.

The top of the page offers some instructions from the Gaim developers as to what we require from a useful bug report. As with most software projects, we need the version of the software the problem is experienced with, the process encountered to reproduce the bug, and (if the bug causes a segfault) a backtrace attained from GDB as shown earlier this chapter. Next are four drop-down boxes: Category, Group, Assigned To, and Priority.

Categorization

The four drop-down boxes allow us to more easily manage bug reports. Category describes what aspect of Gaim the bug affects. This drop-down breaks down Gaim into parts that traditionally are more bug-ridden than others. If the new bug affects one of those categories, the submitter can classify it. Otherwise, the submitter can leave it blank and let a Gaim developer set it.

Group is used to group bugs by version. This allows the Gaim developers easy access to all the bugs in any particular version of Gaim. This field must be set to the correct version because, unlike with Category, we cannot determine from context what version the submitter uses. The remaining two drop-down boxes should not be changed by the submitter.

Assigned To allows us to assign a tracker item to a particular developer, usually the one responsible for the code specified in Category. This is better done by a Gaim developer familiar with the project and each developer's specialties. Priority allows us to judge how important each bug is relative to others. A user submitting a new bug will typically be tempted to give his own bug a higher priority than appropriate. By letting a developer familiar with a large number of bugs handle priorities, they can be more accurate.

Description

Beneath the drop-down boxes are text boxes to type a brief summary and a longer description. The summary should be short but descriptive. When the developers read through a list of bugs, this is the main thing they will look at. "Gaim bug" is a poor summary; "Segfault when sending messages on ICQ" is better. The detailed description that follows should contain enough information for a developer to reproduce the bug, or at least as much information as possible about what causes the bug.

File Attachments

You may want to attach a file to your tracker item. For patches and plug-ins, the use of this is obvious; you need to attach your patch or plug-in. For feature requests, a mocked-up screenshot may be appropriate. For bugs in the GUI, a screenshot of the bug may be helpful.

The biggest mistake in attaching a file to a tracker item is failing to click the Check to Upload and Attach a File check box. A huge number of tracker items are submitted and nearly immediately the submitter updates the item to comment that he forgot to check that check box. To choose a file, click the Browse button or type the path in the provided text box. Then enter a description of the file and hit the Submit button to submit the new tracker item.

Receiving Updates

As the submitter of the item, you will be notified by e-mail when the tracker item is updated, be it assigned to a developer, resolved, or merely commented on. Often a tracker item cannot be resolved without more information. The Gaim developers will add a comment to the item requesting more information; you will receive an e-mail with a link to the tracker item. The page for updating tracker items, seen in Figure 3-15, allows you to change all the aspects you set when creating the item. The description, however, cannot be changed. Instead, you can view a list of comments left about the item, and leave your own comment.

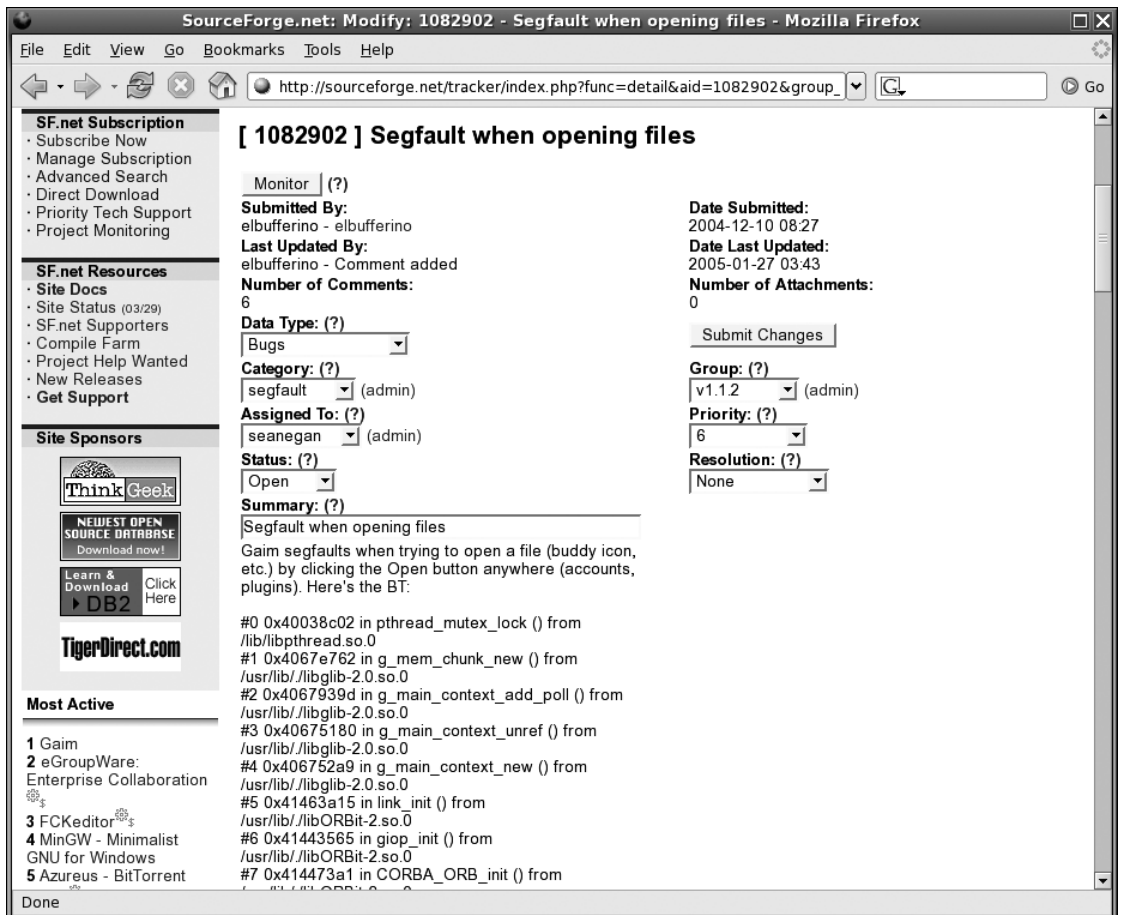


Figure 3-15. The page SourceForge.net uses to update tracker items

Managing Tracker Items

Managing individual tracker items is done from the page shown in Figure 3-15. One considerable difference between creating a new item and modifying an existing one is that modifying an item provides two additional drop-down boxes for Status and Resolution.

The status for all newly created tracker items is Open. Developers can then change it to Deleted, Pending, or Closed. Most commonly, an item will move from Open to Closed. Occasionally, one will change to Pending if the developer was unable to reproduce the error himself, but believes he's fixed it anyway. In this case, the submitter needs to confirm that the bug is actually fixed by changing the status from Pending to Closed. Deleted is rarely used—typically only when a bug report was filed by mistake.

When you close a bug, you provide a resolution for it. This depends on the type of tracker item. Patches are typically Accepted or ejected, bugs are typically Fixed, Duplicate (for issues that have already been reported), Won't Fix (for reports that don't describe actual bugs, but a difference in design opinion), or occasionally Works for Me (for bugs that are entirely irreproducible and, therefore, unfixable).

SourceForge.net offers a lot of other functionality to help you manage your project. The administration interface allows you to provide various levels of access to members of your project and to determine the membership of your project itself. If you are interested in learning more about how to use SourceForge.net's other features, read its documentation at http://sourceforge.net/docman/?group_id=1.

Summary

The growth of open source software has resulted in a large number of very high-quality, useful tools for managing software projects. This, in turn, has opened the door for more open source development. The GNU development tools and SourceForge.net provide everything a potential open source developer needs to get started.

With these tools in hand, in the next chapter you'll begin acquiring the knowledge you need to get involved in Gaim development. I will discuss some of the underlying principles behind Gaim development and some specifics to Gaim itself.