



Red Hat Enterprise Linux 7 Developer Guide

An introduction to application development tools in Red Hat Enterprise Linux 7

Red Hat Subject Matter Experts Dave Brolley

William Cohen

Roland Grunberg

Aldy Hernandez

Karsten Hopp

Jakub Jelinek

Jeff Johnston

Benjamin Kosnik

Chris Moller

Phil Muldoon

Alex Kurtakov

Charley Wang

Development Community

Kent Sebastian

Red Hat Enterprise Linux 7 Developer Guide

An introduction to application development tools in Red Hat Enterprise Linux 7

Dave Brolley
Red Hat Engineering Tools Development
brolley@redhat.com
Profiling

William Cohen
Red Hat Engineering Tools Development
wcohen@redhat.com
Profiling

Roland Grunberg
Red Hat Engineering Tools Development
rgrunber@redhat.com
Eclipse and Eclipse plug-ins

Aldy Hernandez
Red Hat Engineering Tools Development
aldyh@redhat.com
Compiling and Building

Karsten Hopp
Base Operating System Core Services - BRNO
karsten@redhat.com
Compiling

Jakub Jelinek
Red Hat Engineering Tools Development
jakub@redhat.com
Profiling

Jeff Johnston
Red Hat Engineering Tools Development
jjohnstn@redhat.com
Eclipse and Eclipse plug-ins

Benjamin Kosnik
Red Hat Engineering Tools Development
bkoz@redhat.com
Libraries and Runtime Support

Chris Moller
Red Hat Engineering Tools Development
cmoller@redhat.com
Debugging

Phil Muldoon
Red Hat Engineering Tools Development
pmuldoon@redhat.com
Debugging

Alex Kurtakov
Red Hat Engineering Tools Development
akurtako@redhat.com

Legal Notices

Charley Wang
Copyright © 2012 Red Hat, Inc. and others.
Red Hat Engineering Tools Development

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

ken@redhat.com
kenneth.stastian@gmail.com

Profiling

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Development Community

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Jacquelynne East
Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.
Engineering Content Services

jeast@redhat.com
Java ® is a registered trademark of Oracle and/or its affiliates.

Don Domingo
Red Hat is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document describes the different features and utilities that make Red Hat Enterprise Linux 7 an ideal enterprise platform for application development. It focuses on Eclipse as an end-to-end integrated development environment (IDE), but also includes command line tools and other utilities outside Eclipse.

Table of Contents

Chapter 1. Eclipse Development Environment	3
1.1. Starting an Eclipse project	3
1.2. Eclipse User Interface	6
1.3. Editing C/C++ Source Code in Eclipse	15
1.4. Editing Java Source Code in Eclipse	18
1.5. Eclipse RPM Building	19
1.6. Eclipse Documentation	20
Chapter 2. Collaborating	23
2.1. Concurrent Versions System (CVS)	23
2.2. Apache Subversion (SVN)	25
2.3. Git	31
Chapter 3. Libraries and Runtime Support	43
3.1. Version Information	43
3.2. Compatibility	44
3.3. Library and Runtime Details	45
Chapter 4. Compiling and Building	72
4.1. GNU Compiler Collection (GCC)	72
4.2. Distributed Compiling	96
4.3. Autotools	96
4.4. Eclipse Built-in Specfile Editor	98
4.5. CDT in Eclipse	98
4.6. build-id Unique Identification of Binaries	99
4.7. Software Collections and scl-utils	100
Chapter 5. Debugging	102
5.1. ELF Executable Binaries	102
5.2. Installing Debuginfo Packages	103
5.3. GDB	106
5.4. Variable Tracking at Assignments	117
5.5. Python Pretty-Printers	118
5.6. Debugging C/C++ Applications with Eclipse	120
Chapter 6. Profiling	122
6.1. Valgrind	122
6.2. OProfile	125
6.3. SystemTap	129
6.4. Performance Counters for Linux (PCL) Tools and perf	132
6.5. ftrace	135
Chapter 7. Red Hat Developer Toolset	137
7.1. Red Hat Developer Toolset Features	137
7.2. Changes in binutils	138
7.3. Platform Compatibility	143
7.4. Red Hat Developer Toolset References	143
Chapter 8. Red Hat Software Collections	145
8.1. Red Hat Software Collections Features	145
8.2. Platform Compatibility	146
8.3. Red Hat Software Collections Use	146
8.4. Deploy Applications That Use Red Hat Software Collections	147
8.5. Where to Find More Information?	147
Chapter 9. Documentation Tools	148

9.1. Publican	148
9.2. Doxygen	153
Appendix	161
A.1. mallopt	161
malloc_trim	161
malloc_stats	162
Further Information	162
Revision History	163
Index	164

Chapter 1. Eclipse Development Environment

Eclipse is a powerful development environment that provides tools for each phase of the development process. It is integrated into a single, fully configurable user interface for ease of use, featuring a pluggable architecture which allows for extension in a variety of ways.

Eclipse integrates a variety of disparate tools into a unified environment to create a rich development experience. The Valgrind plug-in, for example, allows programmers to perform memory profiling (normally done through the command line) through the Eclipse user interface. This functionality is not exclusive only to Eclipse.

Being a graphical application, Eclipse is a welcome alternative to developers who find the command line interface intimidating or difficult. In addition, Eclipse's built-in **Help** system provides extensive documentation for each integrated feature and tool. This greatly decreases the initial time investment required for new developers to become fluent in its use.

The traditional (that is, mostly command line based) Linux tools suite (**gcc**, **gdb**, etc) and Eclipse offer two distinct approaches to programming. Most traditional Linux tools are far more flexible, subtle, and (in aggregate) more powerful than their Eclipse-based counterparts. These traditional Linux tools, on the other hand, are more difficult to master, and offer more capabilities than are required by most programmers or projects. Eclipse, by contrast, sacrifices some of these benefits in favor of an integrated environment, which in turn is suitable for users who prefer their tools accessible in a single, graphical interface.

1.1. Starting an Eclipse project

Install eclipse with the following command:

```
# yum install eclipse
```

Once installed, Eclipse can be started either by manually executing `/usr/bin/eclipse` or by using the system menu created.

Eclipse stores all project and user files in a designated *workspace*. You can have multiple workspaces and can switch between each one on the fly. However, Eclipse will only be able to load projects from the current active workspace. To switch between active workspaces, navigate to **File > Switch Workspace > /path/to/workspace**. You can also add a new workspace through the **Workspace Launcher** wizard; to open this wizard, navigate to **File > Switch Workspace > Other**.

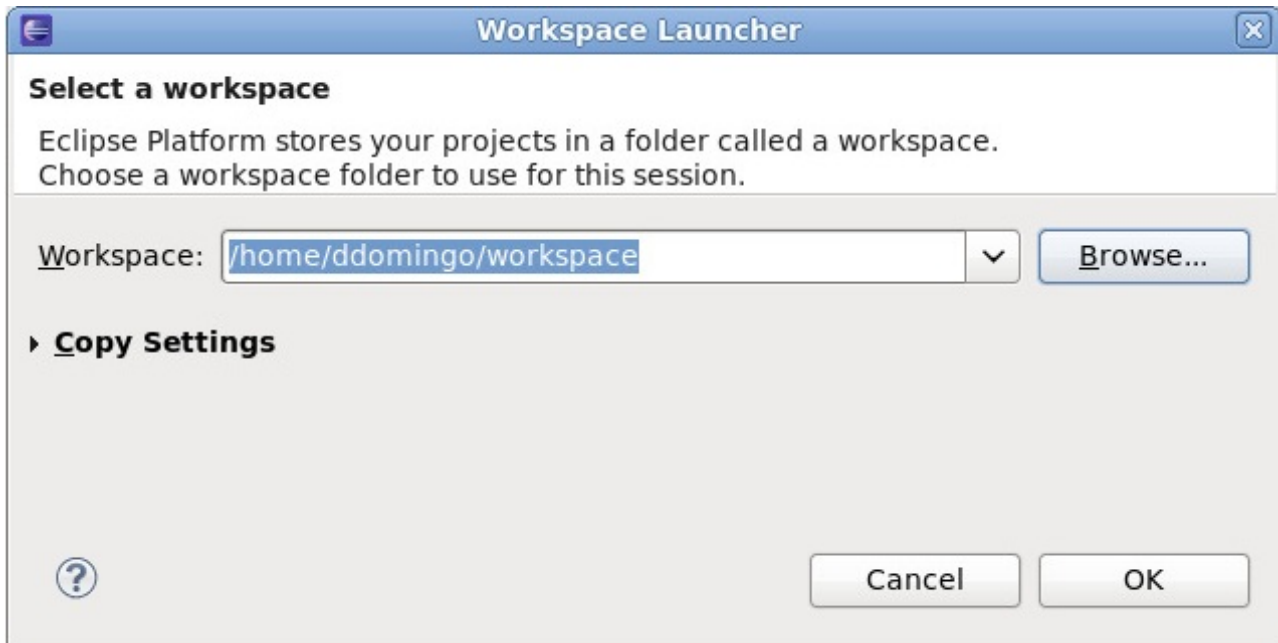


Figure 1.1. Workspace Launcher

For information about configuring workspaces, refer to *Reference > Preferences > Workspace* in the *Workbench User Guide (Help Contents)*.

A project can be imported directly into Eclipse if it contains the necessary Eclipse metafiles. Eclipse uses these files to determine what kind of perspectives, tools, and other user interface configurations to implement.

As such, when attempting to import a project that has never been used on Eclipse, it may be necessary to do so through the **New Project** wizard instead of the **Import** wizard. Doing so will create the necessary Eclipse metafiles for the project, which you can also include when you commit the project.

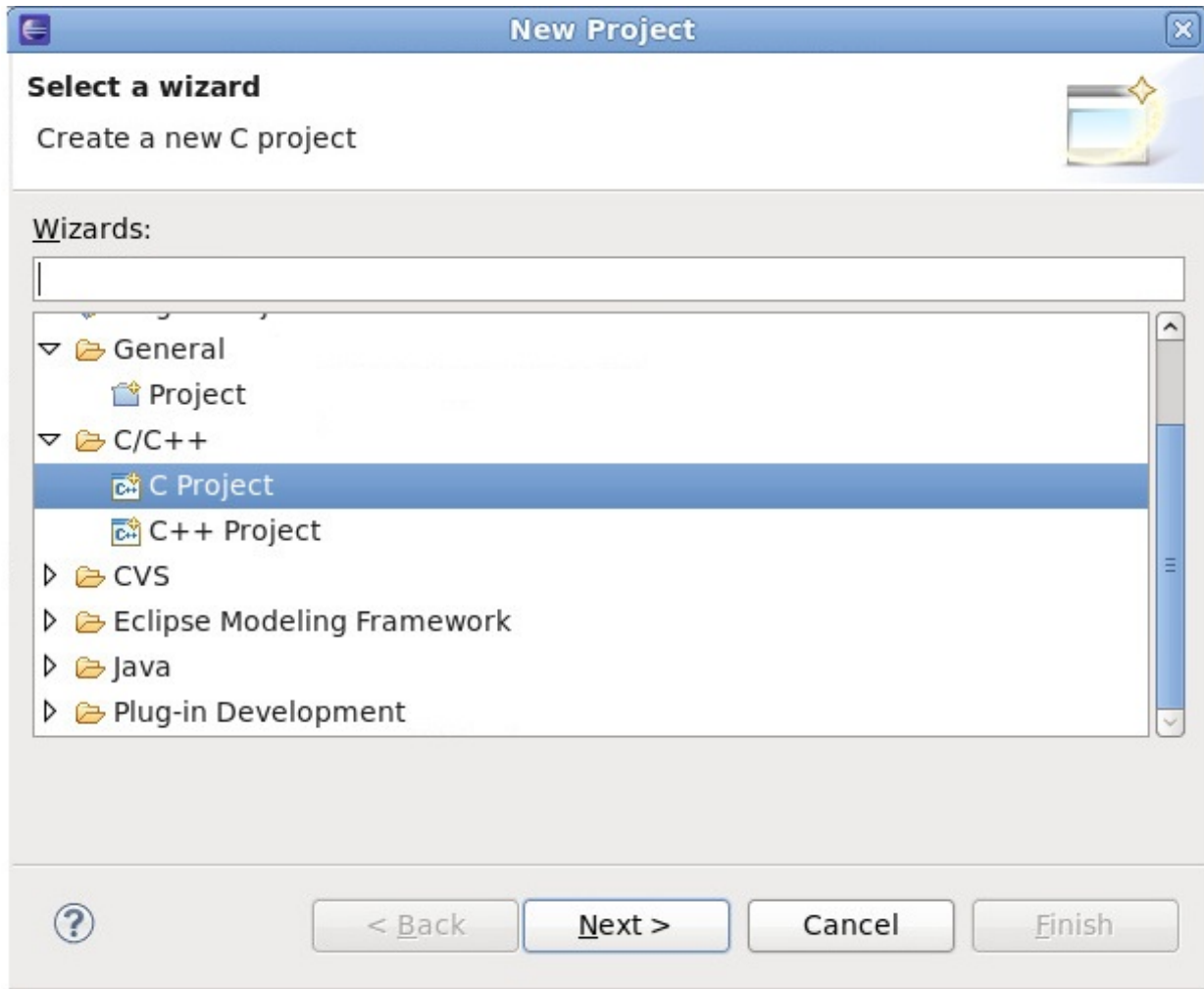


Figure 1.2. New Project Wizard

The **Import** wizard is suitable mostly for projects that were created or previously edited in Eclipse, that is, projects that contain the necessary Eclipse metafiles.

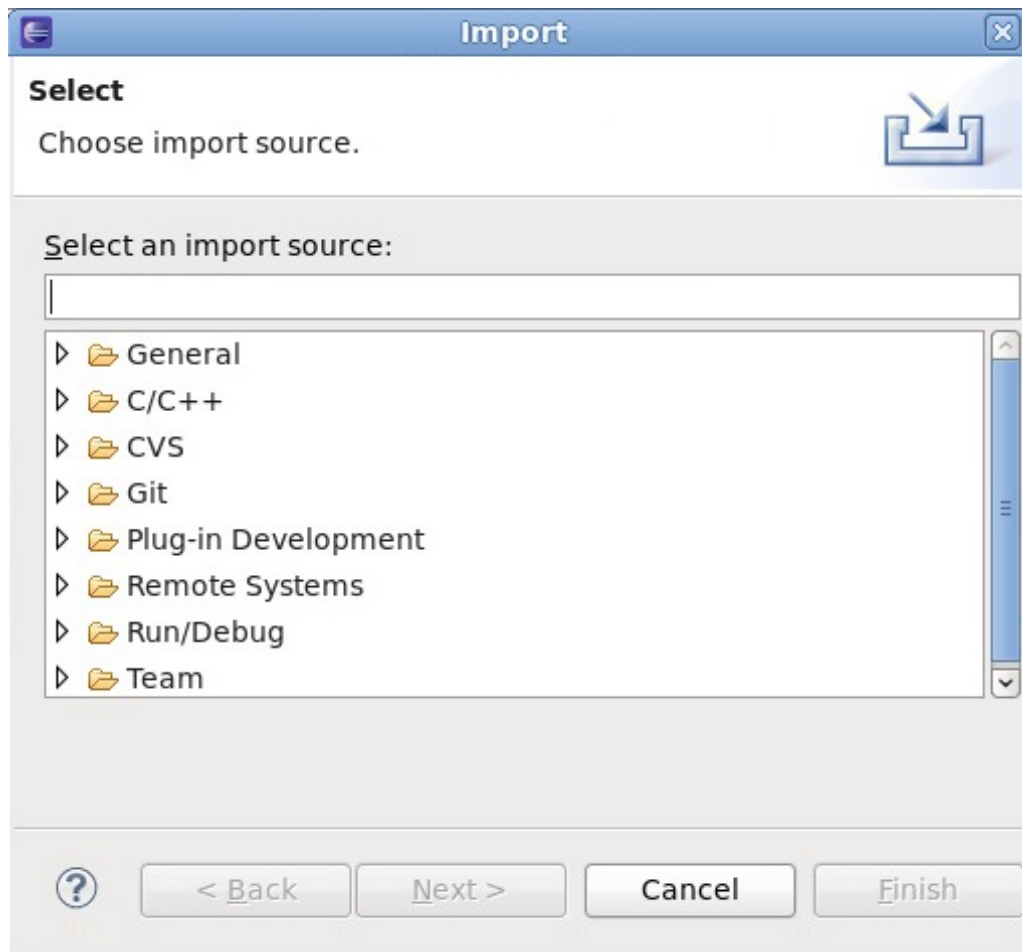


Figure 1.3. Import Wizard

1.2. Eclipse User Interface

The entire user interface in [Figure 1.4, “Eclipse User Interface \(default\)”](#) is referred to as the Eclipse *workbench*. It is generally composed of a code **Editor**, **Project Explorer** window, and several views. All elements in the Eclipse workbench are configurable, and fully documented in the *Workbench User Guide (Help Contents)*. Refer to [Section 1.2.3, “Customize Perspective”](#) for a brief overview on customizing the user interface.

Eclipse features different *perspectives*. A perspective is a set of views and editors most useful to a specific type of task or project; the Eclipse workbench can contain one or more perspectives. [Figure 1.4, “Eclipse User Interface \(default\)”](#) features the default perspective for C/C++.

Eclipse also divides many functions into several classes, housed inside distinct *menu items*. For example, the **Project** menu houses functions relating to compiling/building a project. The **Window** menu contains options for creating and customizing perspectives, menu items, and other user interface elements. For a brief overview of each main menu item, refer to **Reference** → **C/C++ Menubar** in the *C/C++ Development User Guide* or **Reference** → **Menus and Actions** in the *Java Development User Guide*. These are found in the Eclipse help.

The following sections provide a high-level overview of the different elements visible in the default user interface of the Eclipse *integrated development environment (IDE)*.

The Eclipse workbench provides a user interface for many features and tools essential for every phase of the development process. This section provides an overview of Eclipse's primary user interface.

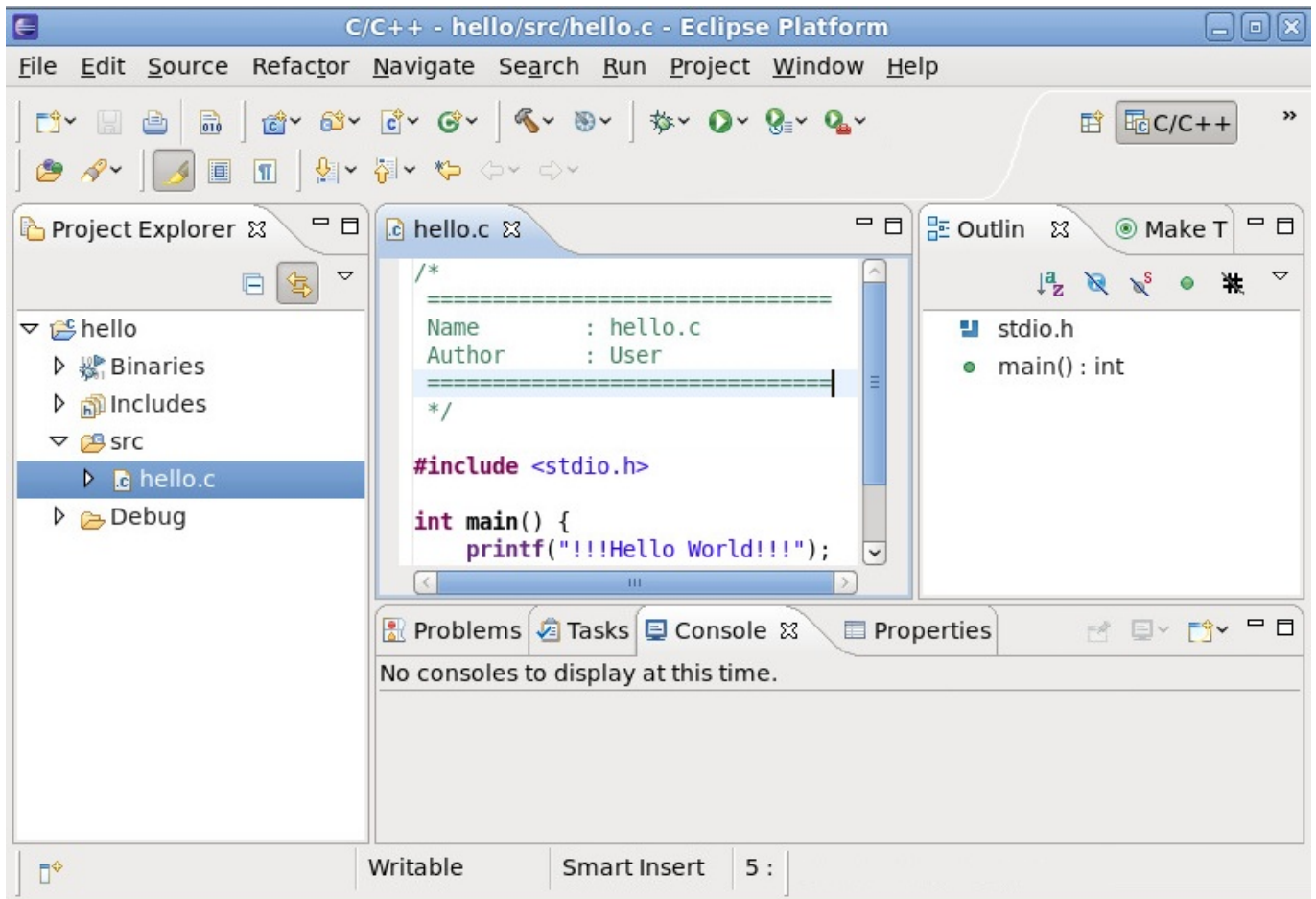


Figure 1.4. Eclipse User Interface (default)

Figure 1.4, “Eclipse User Interface (default)” displays the default workbench for C/C++ projects. To switch between available perspectives in a workbench, press **Ctrl+F8**. For some hints on perspective customization, refer to [Section 1.2.3, “Customize Perspective”](#). The figures that follow describe each basic element visible in the default C/C++ perspective.

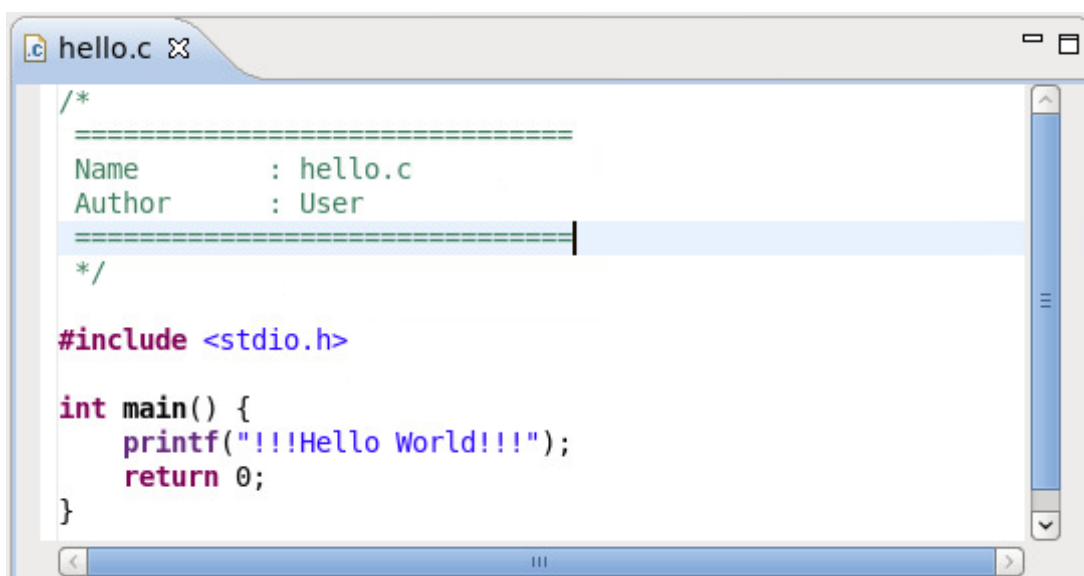


Figure 1.5. Eclipse Editor

The **Editor** is used to write and edit source files. Eclipse can auto-detect and load an appropriate

language editor (for example, C Editor for files ending in `.c`) for most types of source files. To configure the settings for the **Editor**, navigate to **Window > Preferences > language (for example, Java, C++) > Code Style**.

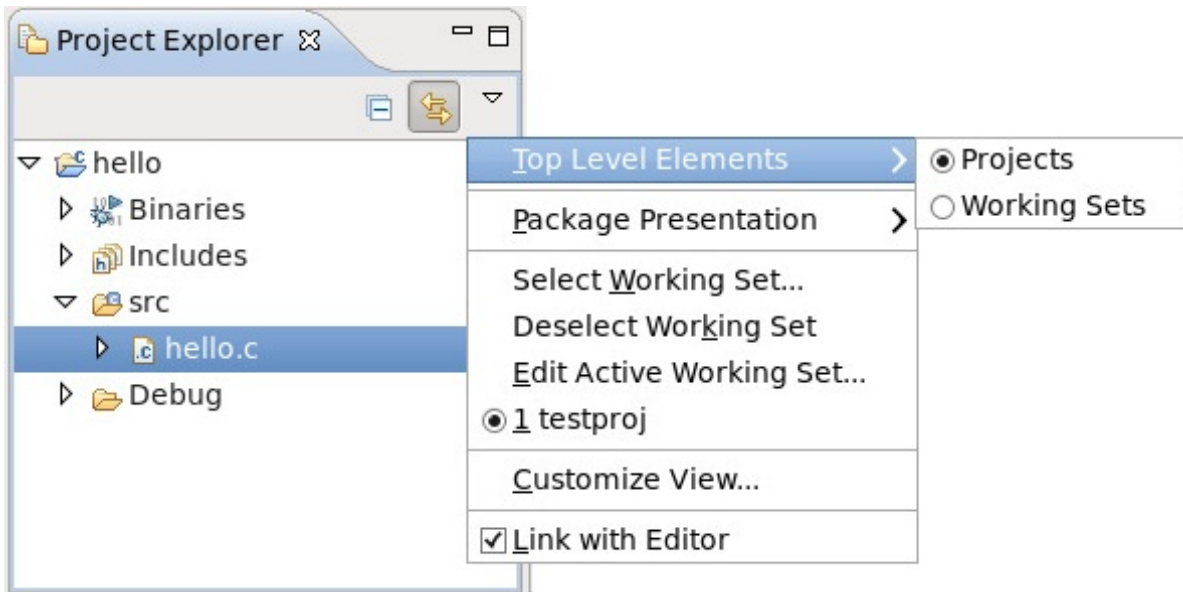


Figure 1.6. Project Explorer

The **Project Explorer View** provides a hierarchical view of all project resources (binaries, source files, etc.). You can open, delete, or otherwise edit any files from this view.

The **View Menu** button in the **Project Explorer View** allows you to configure whether projects or *working sets* are the top-level items in the **Project Explorer View**. A working set is a group of projects arbitrarily classified as a single set; working sets are handy in organizing related or linked projects.

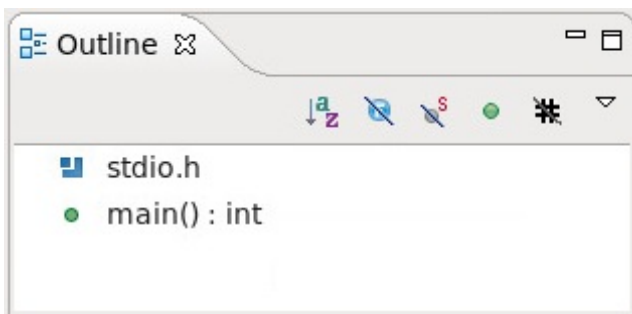


Figure 1.7. Outline Window

The **Outline** window provides a condensed view of the code in a source file. It details different variables, functions, libraries, and other structural elements from the selected file in the **Editor**, all of which are editor-specific.

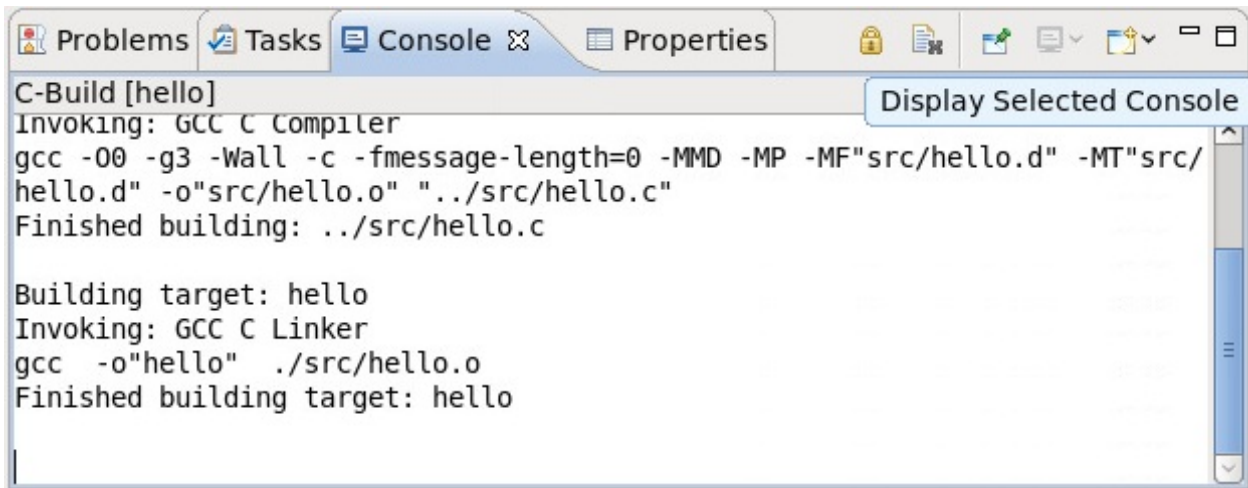


Figure 1.8. Console View

Some functions and plugged-in programs in Eclipse send their output to the **Console** view. This view's **Display Selected Console** button allows you to switch between different consoles.

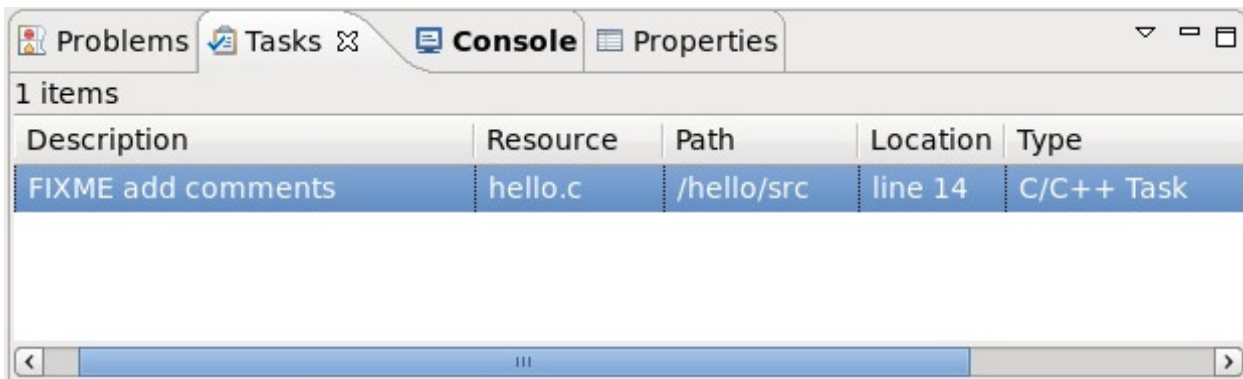


Figure 1.9. Tasks View

The **Tasks** view allows you to track specially-marked reminder comments in the code. This view shows the location of each task comment and allows you to sort them in several ways.



Figure 1.10. Sample of Tracked Comment

Most Eclipse editors track comments marked with `//FIXME` or `//TODO` tags. Tracked comments—that is, *task tags*—are different for source files written in other languages. To add or configure task tags, navigate to **Window > Preferences** and use the keyword **task tags** to display the task tag configuration menus for specific editors/languages.

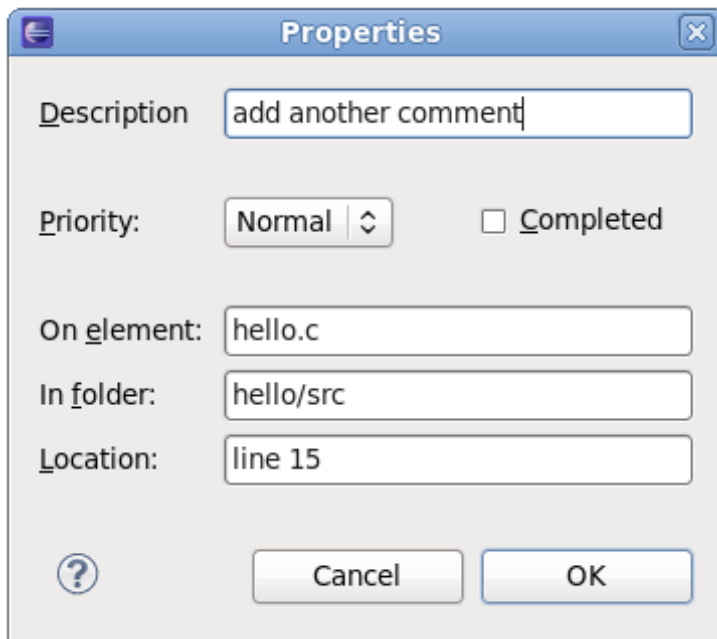


Figure 1.11. Task Properties

Alternatively, you can also use **Edit > Add Task** to open the task **Properties** menu ([Figure 1.11, “Task Properties”](#)). This will allow you to add a task to a specific location in a source file without using a task tag.

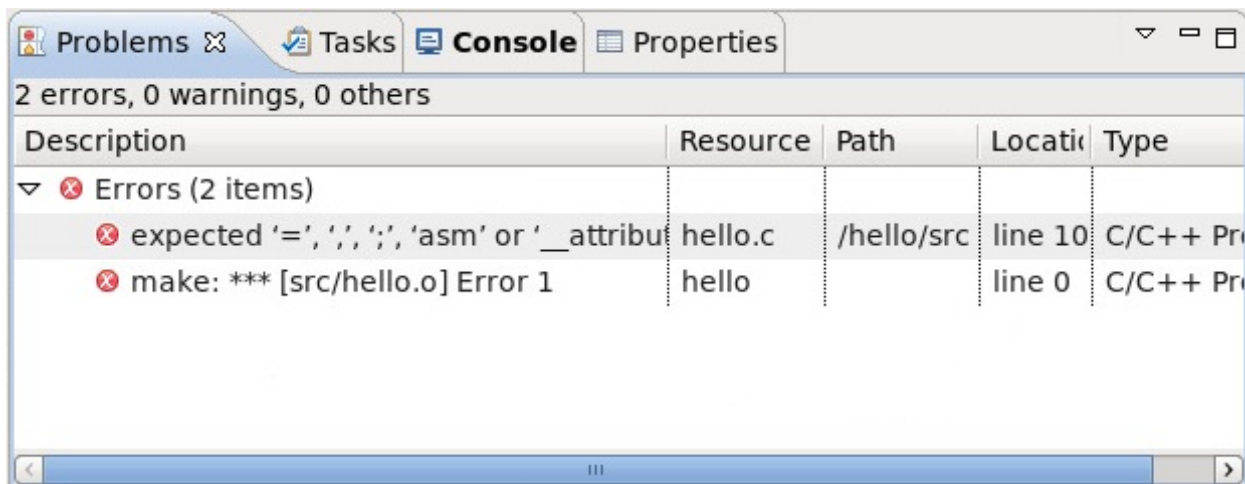


Figure 1.12. Problems View

The **Problems** view displays any errors or warnings that occurred during the execution of specific actions such as builds, cleans, or profile runs. To display a suggested "quick fix" to a specific problem, select it and press **Ctrl+1**.

1.2.1. The Quick Access Menu

One of the most useful Eclipse tips is to use the **quick access** menu. Typing a word in the **quick access** menu will present a list of Views, Commands, Help files and other actions related to that word. To open this menu, press **Ctrl+3**.

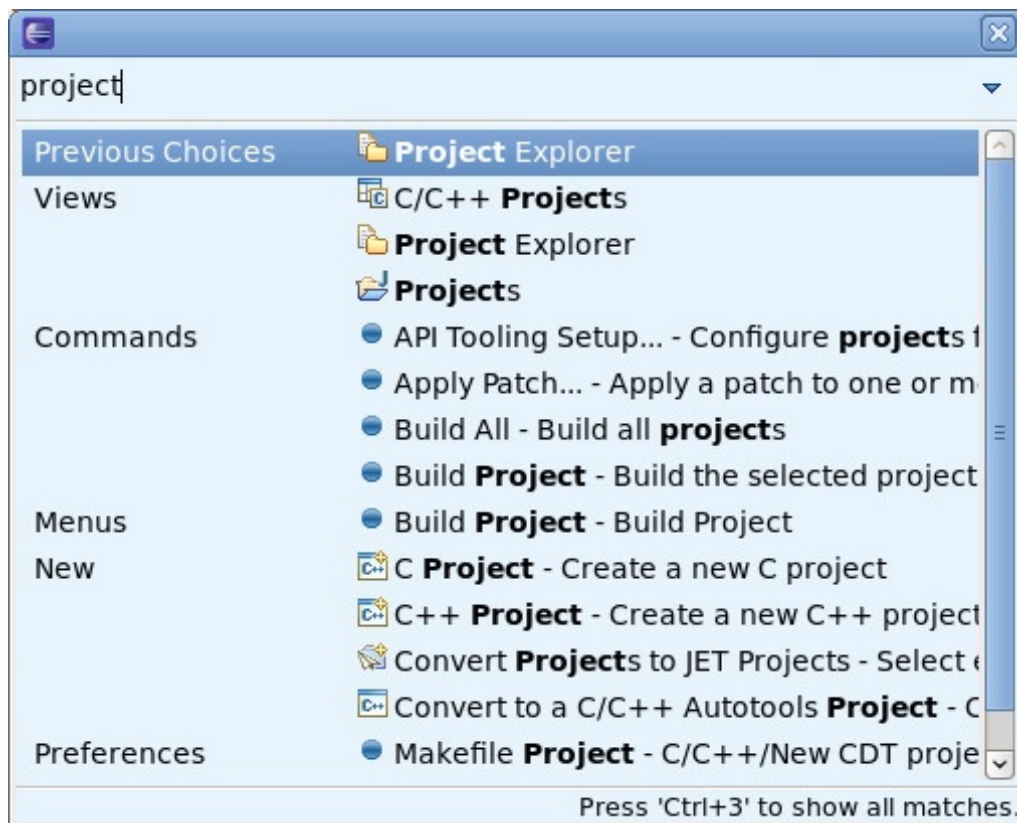


Figure 1.13. Quick Access Menu

In [Figure 1.13, "Quick Access Menu"](#), clicking **Views > Project Explorer** will select the **Project Explorer** window. Click any item from the **Commands**, **Menus**, **New**, or **Preferences** categories to run the selected item. This is similar to navigating to or clicking the respective menu options or taskbar icons. You can also navigate through the **quick access** menu using the arrow keys.

1.2.2. Keyboard Shortcuts

It is also possible to view a complete list of all keyboard shortcut commands; to do so, press **Shift+Ctrl+L**.

Activate Editor	F12
Add Javadoc Comment	Shift+Alt+J
All Instances	Shift+Ctrl+N
Backward History	Alt+Left
Build All	Ctrl+B
Change Method Signature	Shift+Alt+C
Close	Ctrl+W
Close All	Shift+Ctrl+W
Collapse All	Shift+Ctrl+Numpad_Divide

Press "Shift+Ctrl+L" to open the preference page.

Figure 1.14. Keyboard Shortcuts

To configure Eclipse keyboard shortcuts, press **Shift+Ctrl+L** again while the **Keyboard Shortcuts** list is open.

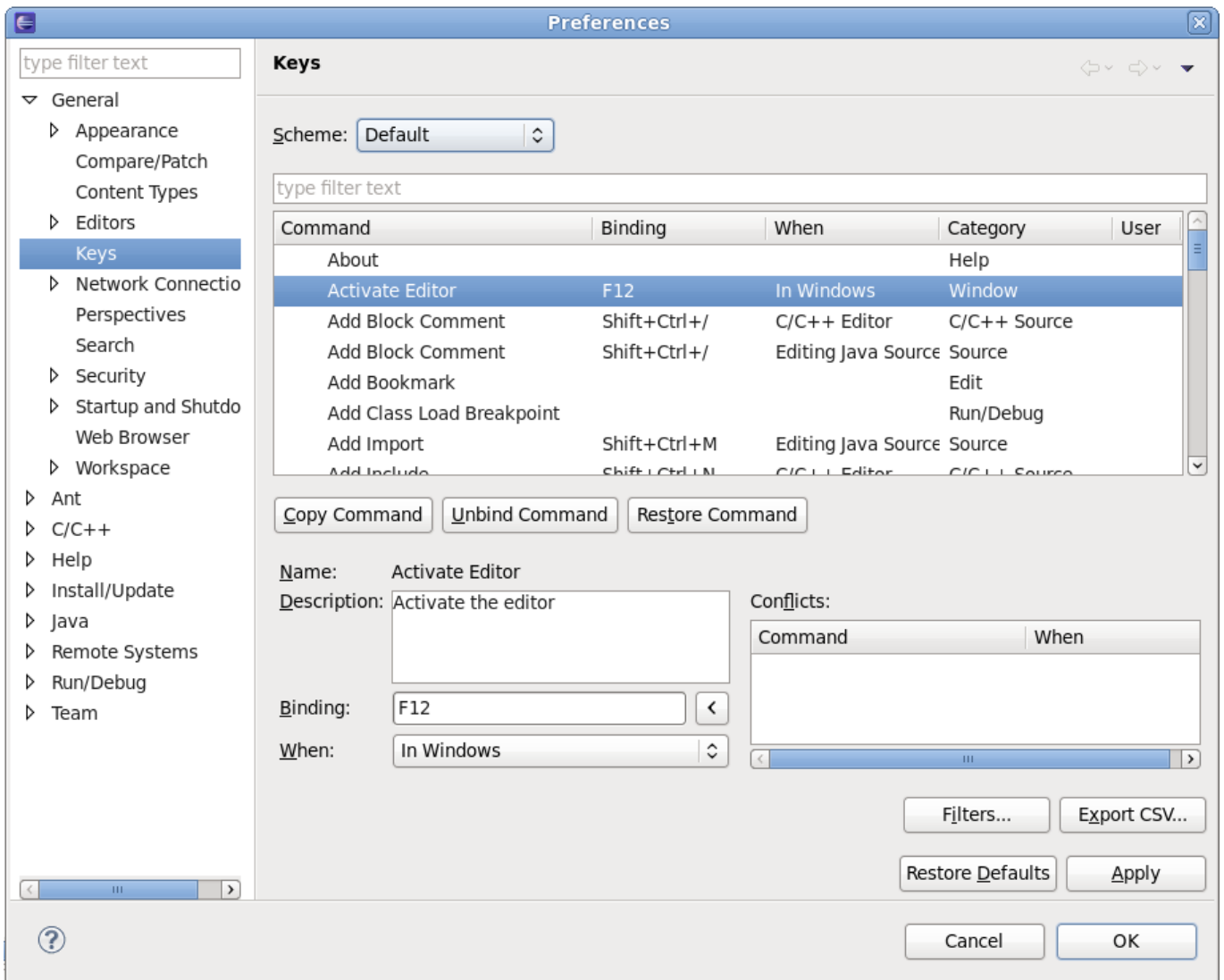


Figure 1.15. Configuring Keyboard Shortcuts

1.2.3. Customize Perspective

To customize the current perspective, navigate to **Window > Customize Perspective**. This opens the **Customize Perspective** menu, allowing the visible tool bars, main menu items, command groups, and shortcuts to be configured.

The location of each view within the workbench can be customized by clicking on a view's title and dragging it to a desired location.

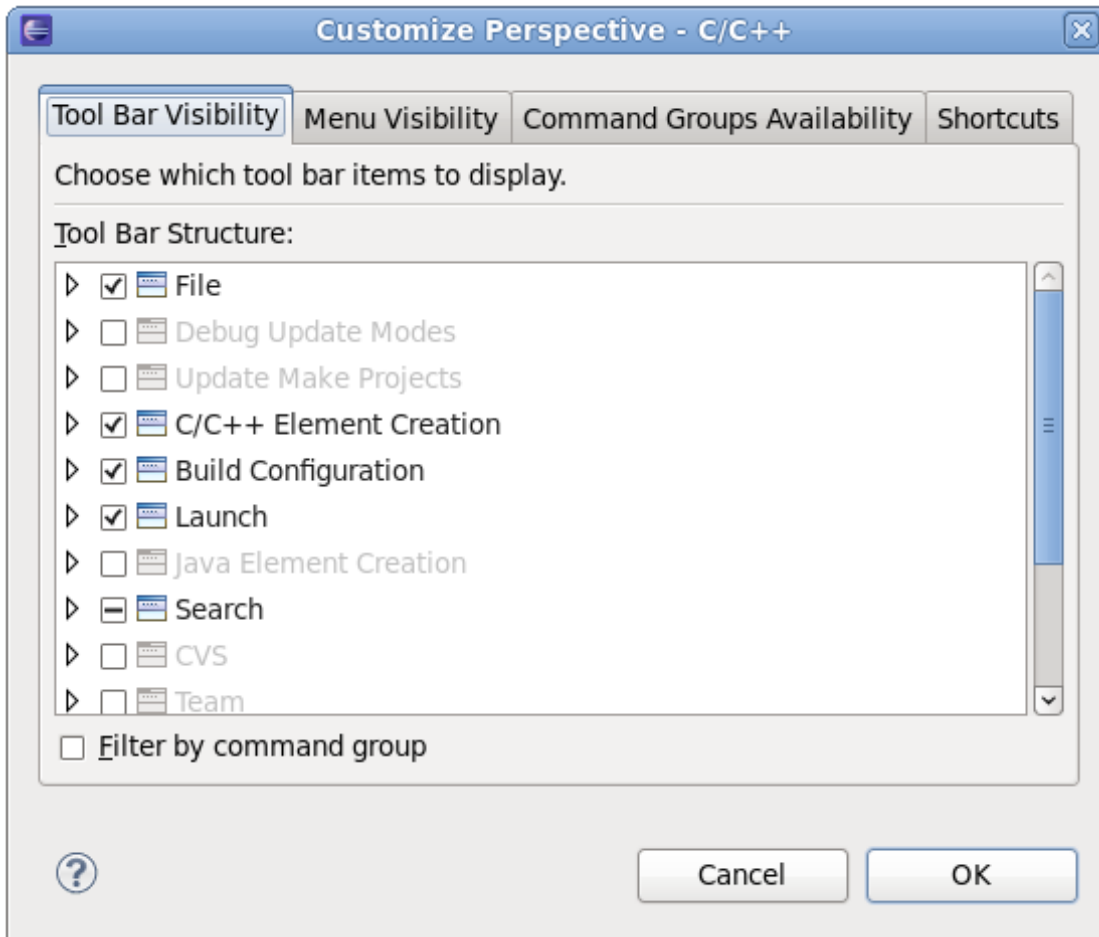


Figure 1.16. Customize Perspective Menu

[Figure 1.16, “Customize Perspective Menu”](#) displays the **Tool Bar Visibility** tab. As the name suggests, this tab allows you to toggle the visibility of the tool bars ([Figure 1.17, “Toolbar”](#)).

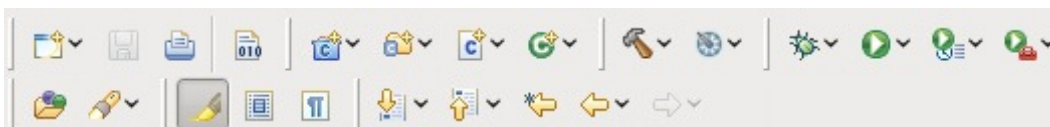


Figure 1.17. Toolbar

The following figures display the other tabs in the **Customize Perspective Menu**:

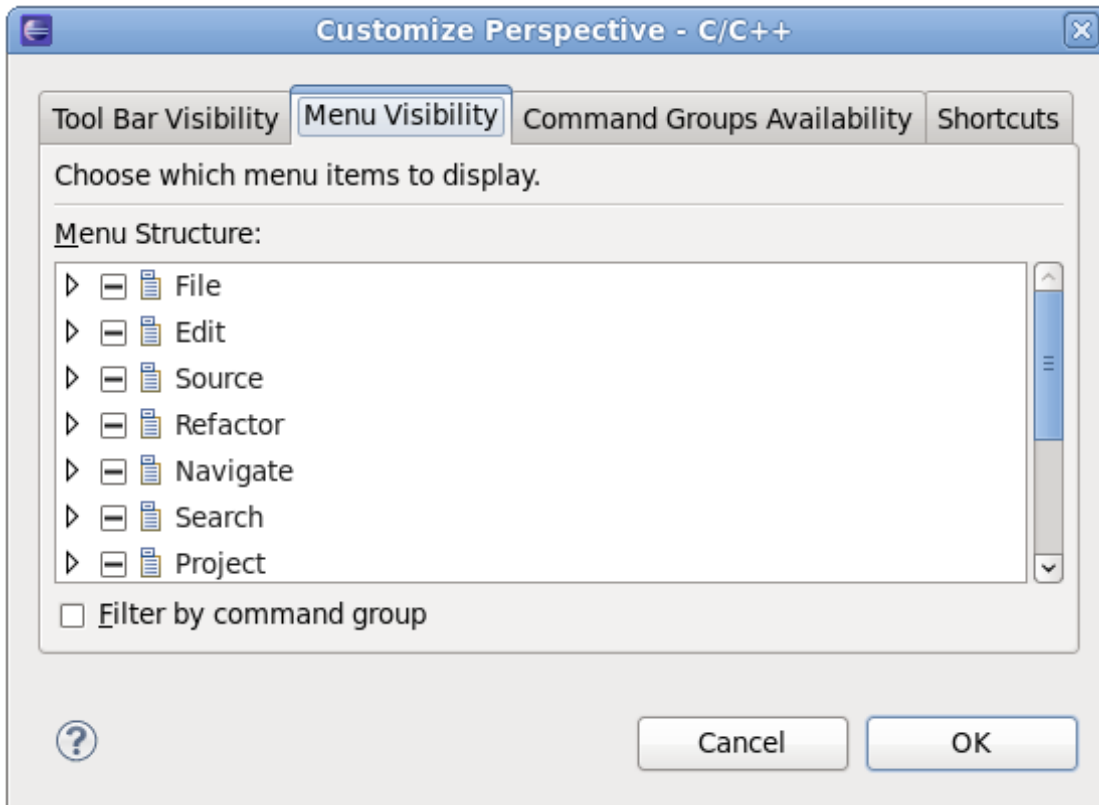


Figure 1.18. Menu Visibility Tab

The **Menu Visibility** tab configures what functions are visible in each main menu item. For a brief overview of each main menu item, refer to *Reference > C/C++ Menubar* in the *C/C++ Development User Guide* or *Reference > Menus and Actions* in the *Java Development User Guide*.

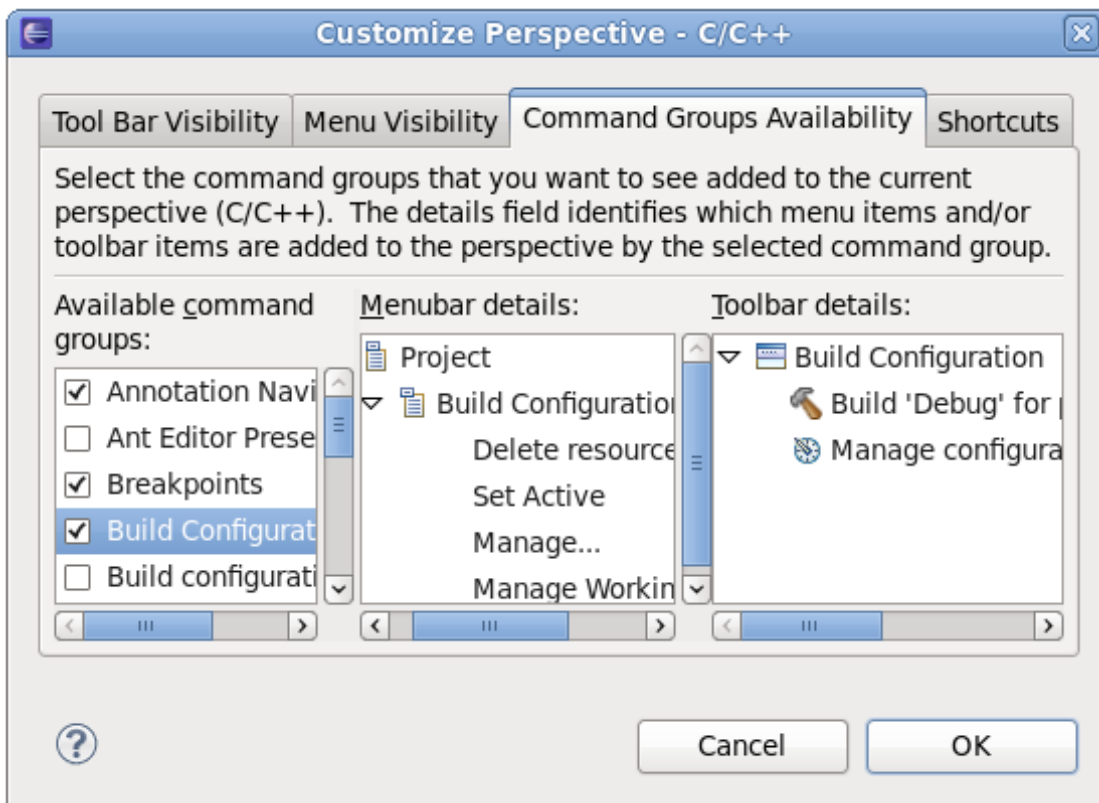


Figure 1.19. Command Group Availability Tab

Command groups add functions or options to the main menu or tool bar area. Use the **Command Group Availability** tab to add or remove a Command group. The **Menubar details** and **Toolbar details** fields display the functions or options added by the Command group to either Main Menu or Toolbar Area, respectively.

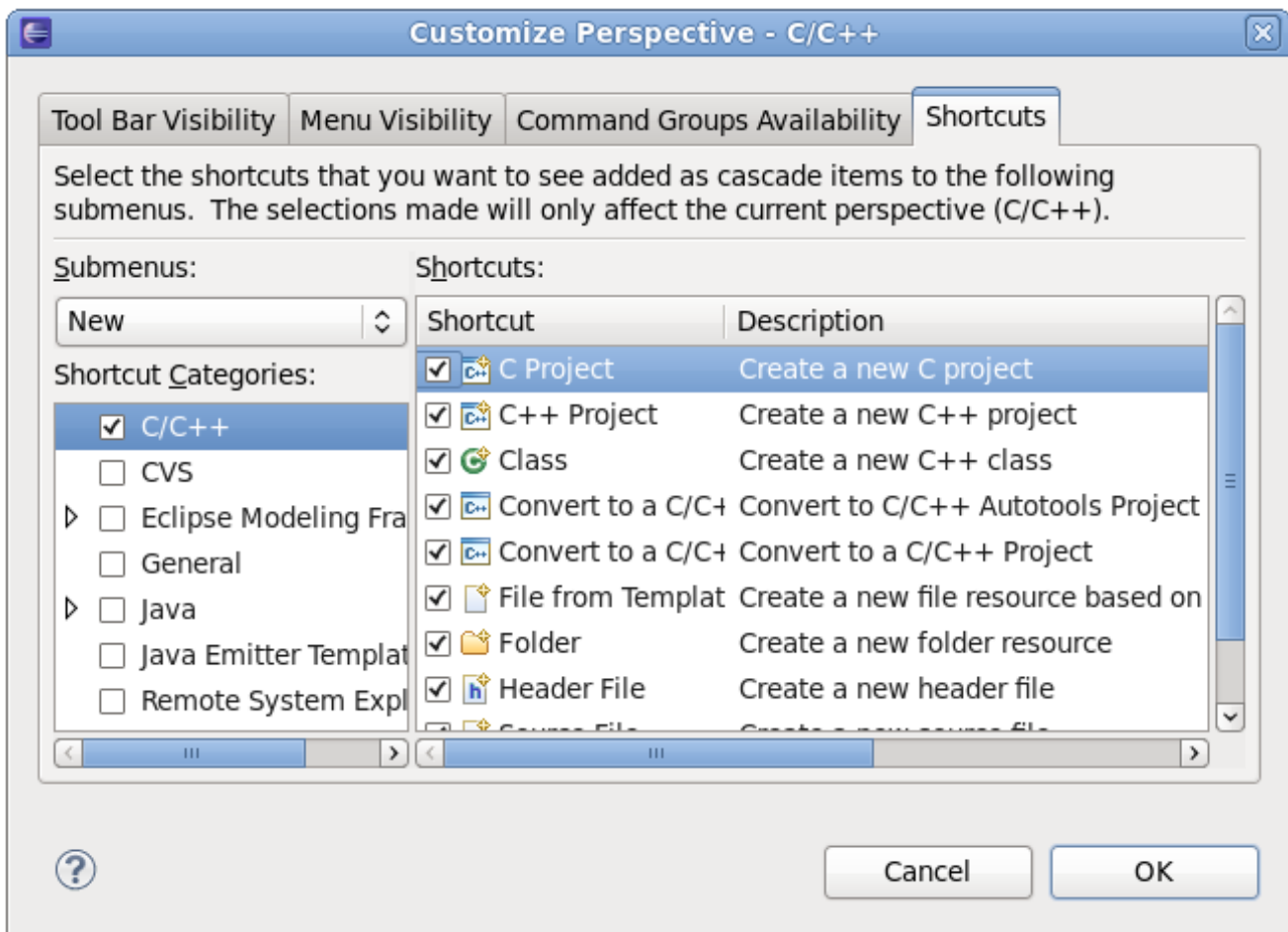


Figure 1.20. Shortcuts Tab

The **Shortcuts** tab configures what menu items are available under the following submenus:

- ▶ **File > New**
- ▶ **Window > Open Perspective**
- ▶ **Window > Show View**

1.3. Editing C/C++ Source Code in Eclipse

Red Hat Enterprise Linux 6 provides Eclipse plug-ins for C/C++ development with the CDT. Specialized editors for C/C++ source code, makefiles, and GNU Autotools files are included. Functionality is also available for running and debugging programs.

The Eclipse text editor supports most of the features expected in a text editor, such as cut, copy, paste, and block selection (**Ctrl+Shift+A**). It also has some relatively unique features, such as the ability to move a selection (**Alt+Up/Down Arrow**).

Of particular interest to C/C++ programmers is the *Content Assist* feature. This feature presents a pop-up window with possible functions, variables, and templates for the current file/location. It is invoked by pressing **Ctrl+Space** while the cursor is at the desired location.

See [Section 1.3.1, “libhover Plug-in”](#) for more information on completion of function calls from libraries.

The Eclipse C/C++ code editor also has error highlighting and refactoring.

Code errors and warnings are annotated with colored wavy underlines. These errors and warnings may be present as code is entered into the editor, or they may be present only after a build has occurred and the compiler output has been transformed into these markers.

The provided refactoring tools include the ability to rename code elements. This change can then be reflected in both uses and declarations of the function.

See [Section 5.6, “Debugging C/C++ Applications with Eclipse”](#), or [Section 4.3.1, “Autotools Plug-in for Eclipse”](#) for more information, or refer to **Concepts** → **Coding aids**, **Concepts** → **Editing C/C++ Files**, and **Tasks** → **Write code** in the *C/C++ Development User Guide*, found in the Help Contents.

1.3.1. libhover Plug-in

The **libhover** plug-in for Eclipse provides plug-and-play hover help support for the GNU C Library and GNU C++ Standard Library. This allows developers to refer to existing documentation on **glibc** and **libstdc++** libraries within the Eclipse IDE in a more seamless and convenient manner via *hover help* and *code completion*.

C++ Language

Documentation for method completion is not supported for C++; only the prototypes from header files are supplied. In addition, the ability to add header files to the source file is not supported for C++ methods.

For C++ library resources, **libhover** has to *index* the file using the CDT indexer. Indexing parses the given file in context of a build; the build context determines where header files come from and how types, macros, and similar items are resolved. To be able to index a C++ source file, **libhover** usually requires an actual build to be performed first, although in some cases it may already know where the header files are located.

A C++ member function name is not enough information to look up its documentation so the **libhover** plug-in may require indexing for C++ sources. C++ allows different classes to have members of the same name, and even within a class, members may have the same name but with different method signatures. This requires the class name and parameter signature of the function to be provided to determine exactly which member is being referenced.

In addition, C++ also has type definitions and templated classes. Such information requires parsing an entire file and its associated **include** files; **libhover** can only do this via indexing.

C Language

For C functions, performing a completion (**Ctrl+Space**) will provide the list of C functions added to the potential sources (for example, typing `prin` and then hitting **Ctrl+Space** will list `printf` as one of the possible completions) and the documentation is viewed in an additional window, for determining exactly what C function is required.

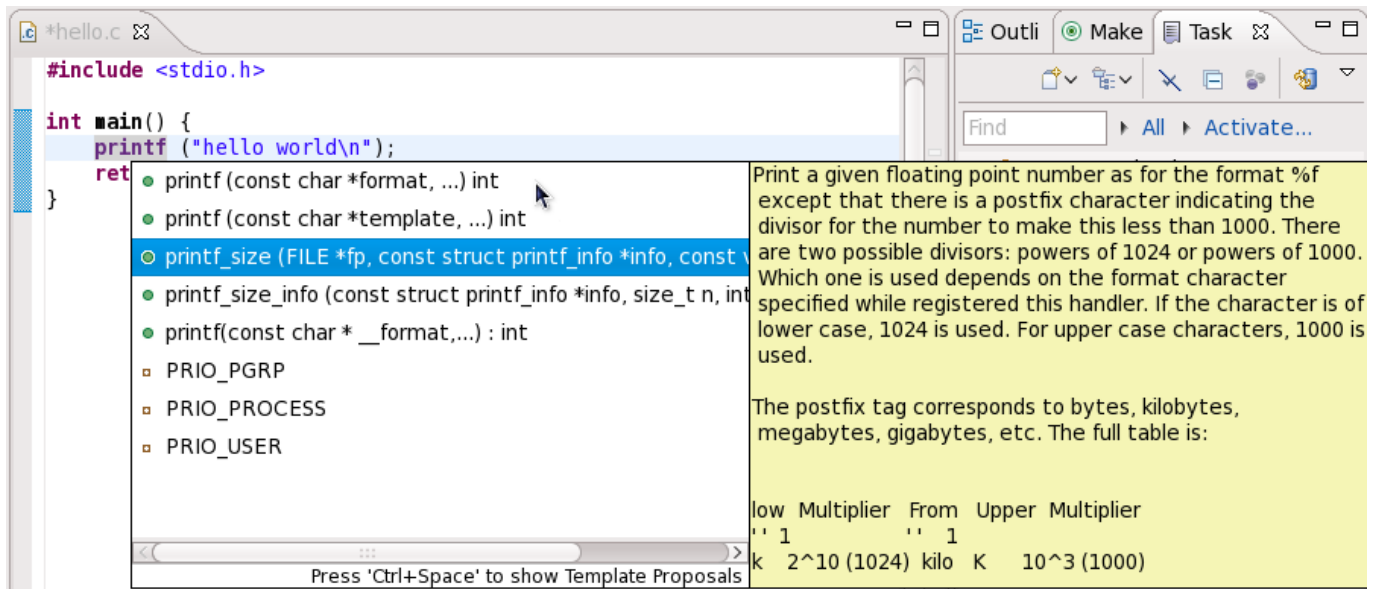


Figure 1.21. Using Code Completion

C functions can be referenced in their documentation by name alone. As such, **libhover** does not have to index C source files in order to provide hover help or code completion. The appropriate header file include statement for a C library function can be automatically added if it is not already present.

Select the C function in the file and use right-click > **Source** > **Add Include** to automatically add the required header files to the source. This can also be done using **Shift+Ctrl+N**.

1.3.1.1. Setup and Usage

Hover help for all installed **libhover** libraries is enabled by default, and it can be disabled per project. To disable or enable hover help for a particular project, right-click the project name and click **Properties**. On the menu that appears, navigate to **C/C++ General** > **Documentation**. Check or uncheck a library in the **Help books** section to enable or disable hover help for that particular library.

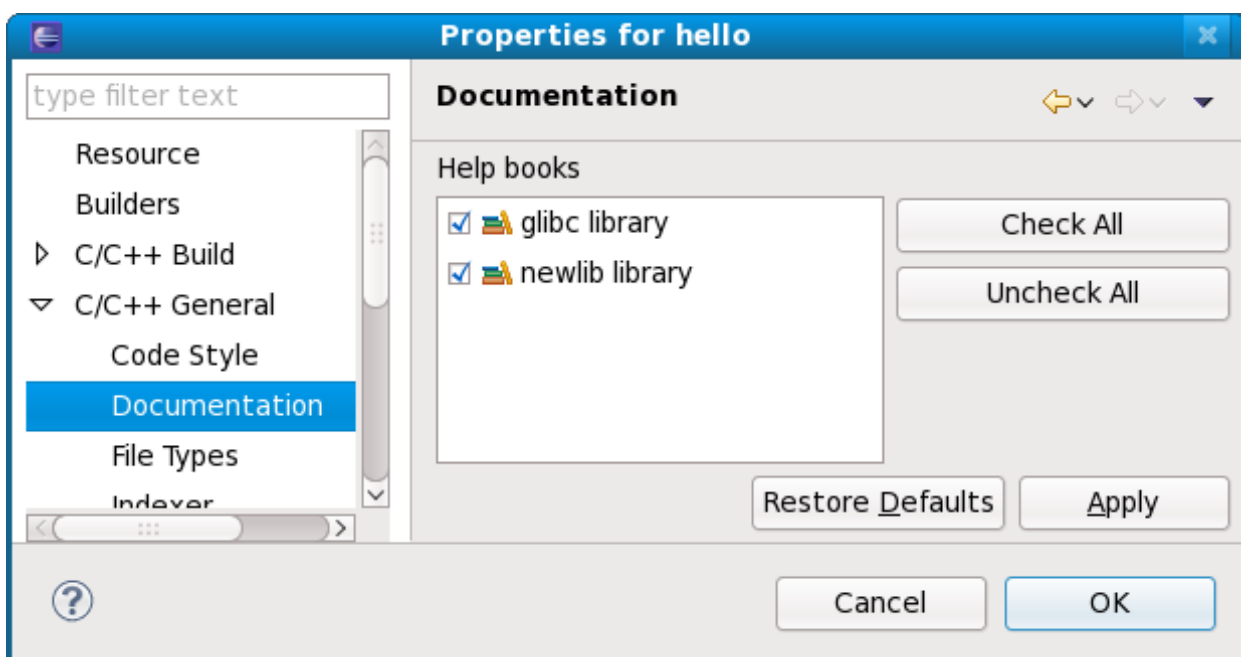


Figure 1.22. Enabling/Disabling Hover Help

Disabling hover help from a particular library may be preferable, particularly if multiple **libhover** libraries overlap in functionality. For example, if a **libhover** plug-in for a C library were manually installed, such as the **newlib** C library (note that **newlib** C library plug-in is not provided in Red Hat Enterprise Linux 6). The hover help would contain C functions whose names overlap with those in the GNU C library (provided by default). A user would not want both of these hover helps active at once, so disabling one would be practical.

When multiple **libhover** libraries are enabled and there exists a functional overlap between libraries, the Help content for the function from the *first* listed library in the **Help books** section will appear in hover help (that is, in [Figure 1.22, “Enabling/Disabling Hover Help”](#), **glibc**). For code completion, **libhover** will offer all possible alternatives from all enabled **libhover** libraries.

To use hover help, hover the mouse over a function name or member function name in the **C/C++ Editor**. After a short time, no more than a few seconds, **libhover** will display library documentation on the selected C function or C++ member function.

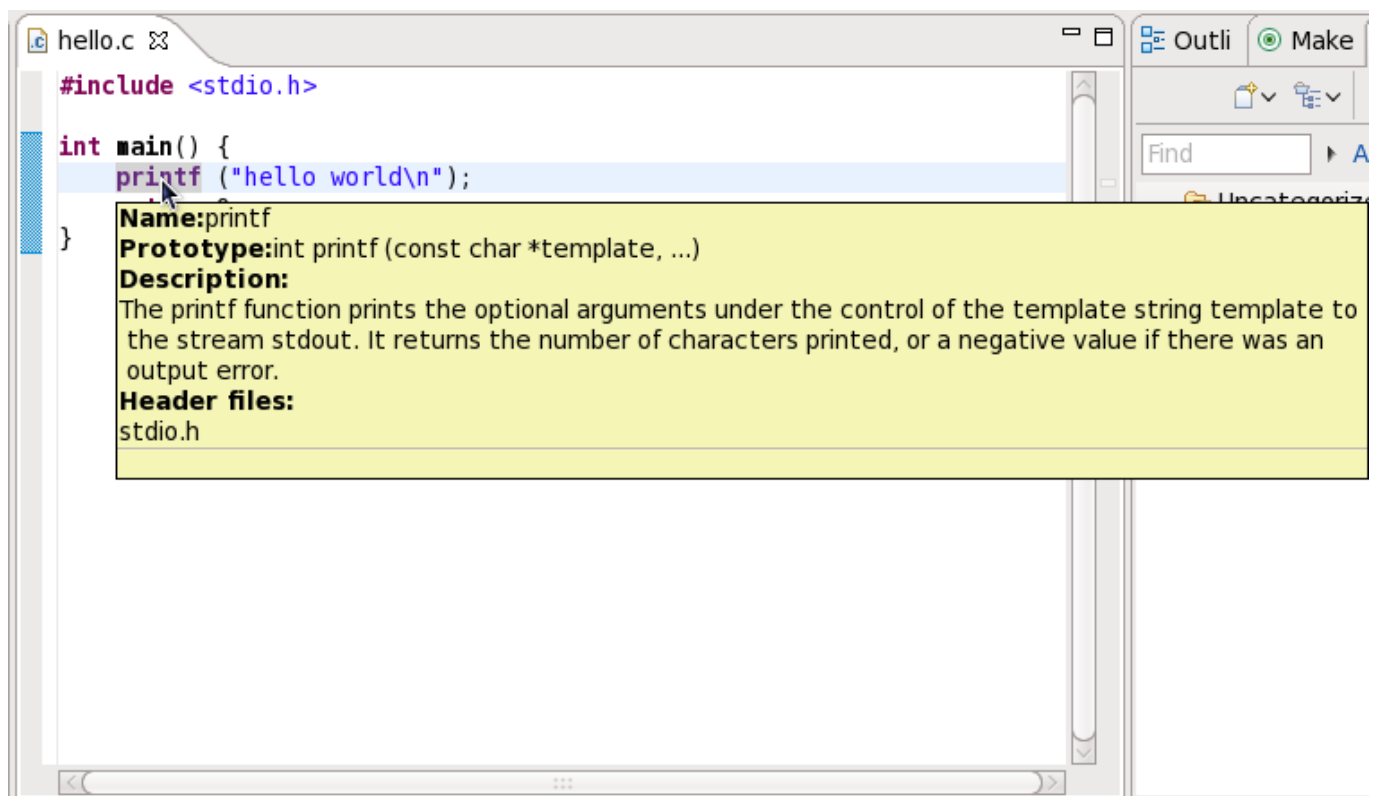


Figure 1.23. Using Hover Help

1.4. Editing Java Source Code in Eclipse

Red Hat Enterprise Linux 6 provides Eclipse plug-ins for Java (Java SE) development with the JDT. Specialized editors for Java source code as well as for ant **build.xml** files are included. Functionality is also available for running and debugging programs.

Eclipse provides a fully-featured interactive development environment for Java developers.

New Project Wizard

Eclipse's **New Project Wizard** performs most of the boilerplate setup required to start a Java project. This allows the user to select and customize various options, such as which Java Runtime Environment to use, or the preferred project file tree layout.

Follow the same procedure to export an existing project, and when prompted for a location, enter the existing project's location instead.

For more information regarding setting up a new Java project, refer to **Help > Help Contents > Java Development > Getting Started > Basic Tutorial > Creating Your First Java Project**.

Content Assistance

The Eclipse Java Development Environment (JDT) increases productivity and reduces errors by providing a rich set of content assistance features, usually invoked by pressing **Ctrl + Space**. This includes completion for method names in your code or in libraries, inserting parameter names in Javadoc, and filling in parameters when calling a method. This is fully customizable, allowing the options of suppressing certain suggestions or adding custom code templates to be filled in while writing code.

For an overview of these features, refer to **Help > Help Contents > Java Development User Guide > Tips and Tricks**.

Code Formatting

Code formatting, accessed by pressing **Ctrl + Shift + F**, is another useful feature present in the JDT. The formatting settings can be changed by navigating to **Window > Preferences > Java > Code Styler > Formatter** where there is the option of using a set of installed formatting profiles, or creating a new one to fit the style of the project.

Debugging Features

The JDT also comes with several debugging features. Create breakpoints by double-clicking on the left hand margin at the desired line of code. When the debugger is run, the program will stop at that line of code which is useful in detecting the location of errors.

The Debug Perspective, configured the first time the debugger is run, is a different layout that makes views related to debugging more prominent. For example, the **Expressions** view allows evaluation of Java expressions in the context of the current frame.

The views that make up the Debug Perspective, like all views, are accessed through **Window > Show View** and you do not have to be debugging to access these views.

While debugging, hover over variables to view their values or use the **Variables** view. Using the Debug view, it is possible to control the execution of programs and explore the various frames on the stack.

For more information on debugging in the JDT, refer to **Help > Help Contents > Java Development > Getting Started > Basic Tutorial > Debugging Your Programs**.

JDT Features

The JDT is highly customizable and comes with an extensive feature list which can be viewed through the settings in **Window > Preferences > Java**, and through the Java settings in **Project > Properties**. For detailed documentation of the JDT and its features, refer to the *Java Development User Guide* found in **Help > Help Contents > Java Development User Guide**.

1.5. Eclipse RPM Building

The Specfile Editor Plug-in for Eclipse provides useful features to help developers manage **.spec** files. This plug-in allows users to leverage several Eclipse GUI features in editing **.spec** files, such as auto-completion, highlighting, file hyperlinks, and folding.

In addition, the Specfile Editor Plug-in also integrates the **rpm1int** tool into the Eclipse interface. **rpm1int** is a command line tool that helps developers detect common RPM package errors. The richer visualization offered by the Eclipse interface helps developers quickly detect, view, and correct mistakes reported by **rpm1int**.

The Eclipse **.spec** file editor plug-in also supports building RPM files from RPM projects. This feature can be used by employing an export wizard (**Import** → **RPM** → **Source/Binary RPM**) allowing the selection of whether a source RPM (**src.rpm**), binary RPM, or both, are required.

Build output is in the Eclipse Console view. For a limited number of build failures, there is hyperlinking support. That is, certain parts of the build failure are changed to be a hyperlink (**Ctrl+Click**) in the Eclipse Console view, which then points the user to the actual line in the **.spec** file that is causing the problem.

Also of note is the wizard for importing source RPM (**.src.rpm**) files, found in **Import** → **RPM** → **Source RPM**). This allows the user to easily start with no configuration required, in case a source RPM has already been created. This project is then ready for editing the spec file and building (exporting) to source/binary RPMs.

For further details, refer to the **Specfile Editor User Guide** → **Import src.rpm and export rpm and src.rpm** section in the *Specfile Editor User Guide* in Help Contents.

1.6. Eclipse Documentation

Eclipse features a comprehensive internal help library that covers nearly every facet of the Integrated Development Environment (IDE). Every Eclipse documentation plug-in installs its content to this library, where it is indexed. To access this library, use the **Help** menu.

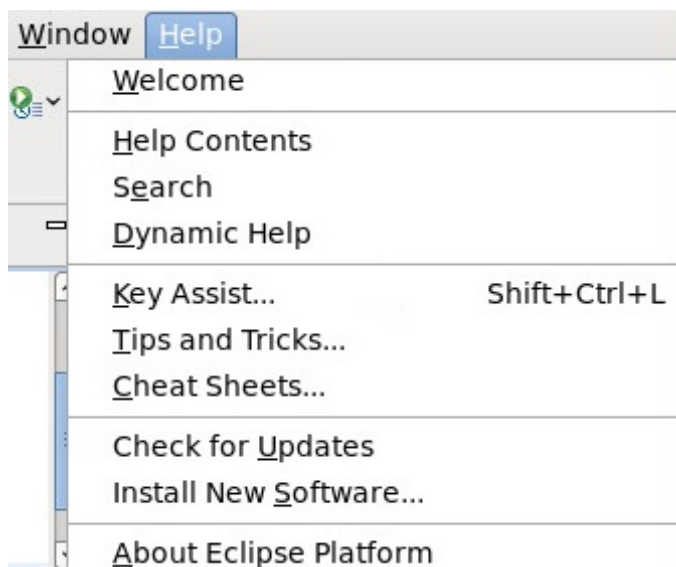


Figure 1.24. Help

To open the main **Help** menu, navigate to **Help** > **Help Contents**. The **Help** menu displays all the available content provided by installed documentation plug-ins in the **Contents** field.

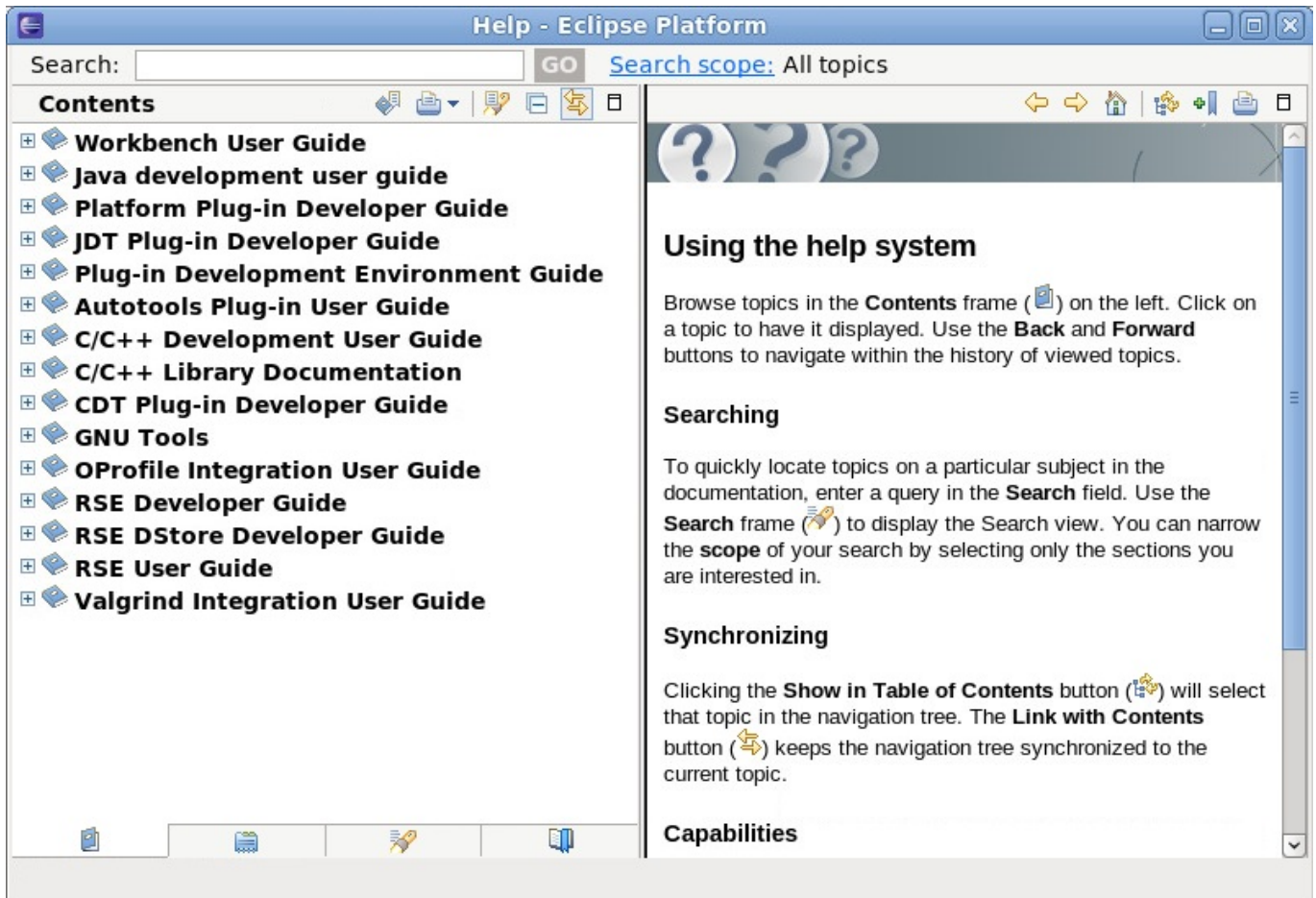


Figure 1.25. Help Menu

The tabs at the bottom of the **Contents** field provides different options for accessing Eclipse documentation. You can navigate through each "book" by section/header or by searching via the **Search** field. You can also bookmark sections in each book and access them through the **Bookmarks** tab.

The *Workbench User Guide* documents all facets of the Eclipse user interface extensively. It contains very low-level information on the Eclipse workbench, perspectives, and different concepts useful in understanding how Eclipse works. The *Workbench User Guide* is an ideal resource for users with little to intermediate experience with Eclipse or IDEs in general. This documentation plug-in is installed by default.

The Eclipse help system also includes a *dynamic help* feature. This feature opens a new window in the workbench that displays documentation relating to a selected interface element. To activate dynamic help, navigate to **Help > Dynamic Help**.

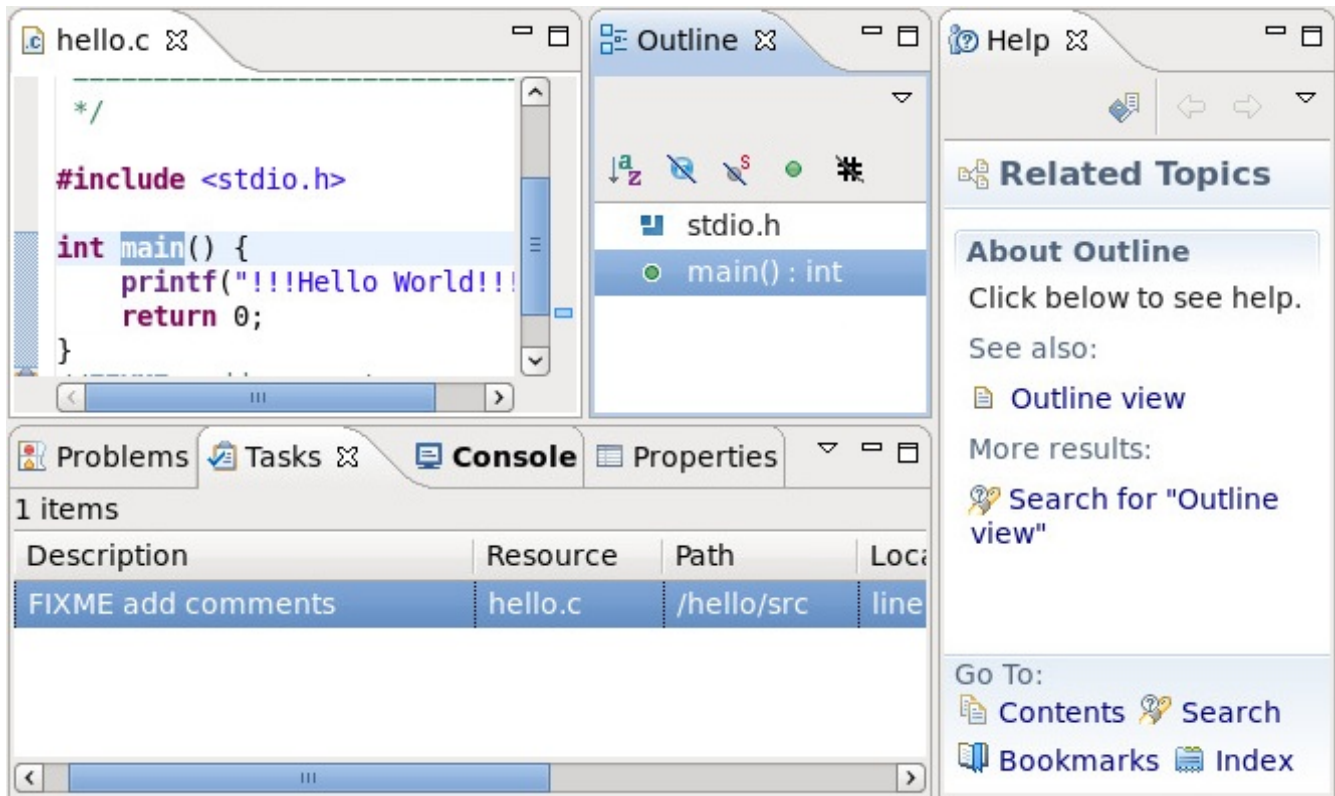


Figure 1.26. Dynamic Help

The rightmost window in [Figure 1.26, "Dynamic Help"](#) displays help topics related to the **Outline** view, which is the selected user interface element.

Chapter 2. Collaborating

Effective revision control is essential to all multi-developer projects. It allows all developers in a team to create, review, revise, and document code in a systematic and orderly manner. Red Hat Enterprise Linux 6 supports three of the most popular open source revision control systems: CVS, SVN, and Git. The tools for these revision control systems provide access to a wide range of publicly available open source code, as well as the capability to set up individual internal code repositories.

The following section provides a brief overview and references to relevant documentation for each tool.

2.1. Concurrent Versions System (CVS)

Concurrent Versions System (CVS) is a centralized version control system based on RCS format with a client-server architecture. It was the first version control system and the predecessor for Subversion (SVN).

2.1.1. CVS Overview

This section discusses the various elements of CVS, both the good and the bad.

CVS was developed when network connectivity was unreliable and would often drop out. This meant that if several files were committed at once and the network dropped out, the commit would fail. This can still occur now if a network is unreliable but is less common with modern networking infrastructure. If it happens, the CVS administrator has two options to resolve the problem. The first is to use the **admin** command to remove stall locked files and back out the changed files. The second option is to reissue the commit command.

CVS uses one central location for making back-ups, which is useful for an unstable network. It allows the enforcement of a commit policy through manually prepared triggers (automated tests, builds, Access Control Lists (ACLs), integration with a bug tracking system) due to centralized architecture.

To create more detailed commits to the backup, CVS can also expand keywords that are marked by the at-sign (@) to record commit details (committer name, commit message, commit time, for example) into a committed file.

In order to keep track of these commits, CVS uses a server to track the changes for each file separately and in reverse time order. By doing so, the latest version is stored directly and can be retrieved quickly, where older versions must be recomputed by the server. Each changed, committed file is tracked separately with an independent revision identifier. This can make it difficult to discover which files have been changed by the commit when multiple changed files are committed. To counter this, users should tag the repository state to refer back and view the changes when required.

The CVS repository can be accessed by two methods. If the repository is on the same machine as the client (:local: access method) then the client spawns the server on its behalf. If the repository is on a remote machine, the server can be started with rsh/SSH (**CVS_RHS** environment variable) by a client or by an inet daemon (/etc/xinetd.d/cvs) and different authentication methods (:gserver: access method integrates Kerberos authentication, for example) can be used.

Finally, for security a client-server approach is used with CVS. This means that the client is dependent on connectivity to the server and cannot perform any operation (committing, or reading the commit log) without permission to access the server.

2.1.2. Typical Scenario

This is a sequence of commands demonstrating CVS repository creation in the **\$CVSROOT** directory

(using an absolute path to signal **:local:** access method), importing sources from **\$SOURCES**, checking them out from the repository into **\$WORKDIR**, modifying some files, and committing the changes.

Procedure 2.1. Using CVS

1. Initialize CVS storage.

```
$ mkdir "$CVSRROOT"  
$ cvs -d "$CVSRROOT" init
```

This creates the **CVSRROOT** subdirectory under **\$CVSRROOT** with repositories configuration.

2. Import code from **\$SOURCES** directory into CVS as **\$REPOSITORY**, tagged with **\$VENDOR_TAG** and **\$RELEASE_TAG** with a commit **\$MESSAGE**.

```
$ cd "$SOURCES"  
$ cvs -d "$CVSRROOT" import -m "$MESSAGE" "$REPOSITORY" \  
"$VENDOR_TAG" "$RELEASE_TAG"
```

The **\$SOURCES** content should be imported into CVS under **\$CVSRROOT/\$REPOSITORY**. It is possible to have more repositories in one CVS storage, though this example just uses the one. The **\$VENDOR_TAG** and **\$RELEASE_TAG** are tags for implicit repository branches.

3. Different developers can now check the code out into **\$WORKDIR**.

```
$ cd "$WORKDIR"  
$ cvs -d "$CVSRROOT" checkout "$REPOSITORY"
```



Warning

Do not check out into the original **\$SOURCES**. This could cause data corruption on the client side and CVS will print errors on various CVS invocations.

4. The latest version of the CVS repository has been transferred into the **\$REPOSITORY** subdirectory. The developer can also check out multiple repositories from one server.

```
$ cd $REPOSITORY
```

5. To schedule adding a new **\$FILE** use:

```
$ vi "$FILE"  
$ cvs add "$FILE"
```

6. The developer can modify an **\$EXISTING_FILE**.

```
$ vi "$EXISTING_FILE"
```

7. Finally, the developer can commit these changes with a **\$COMMIT_MESSAGE**.

```
$ cvs commit -m "$COMMIT_MESSAGE"
```

It is possible to export the **\$CVSROOT** value as a **CVSROOT** environment variable and the **cv**s tool will respect it. This can free the developer from repetitively supplying the **-d "\$CVSROOT"** option. The value is stored in the CVS helper subdirectory at initial check-out, and the CVS tool takes the value from there automatically.

2.1.3. CVS Documentation

The CVS manual page can be accessed with **man cvs**.

There is also a local FAQ page located in **/usr/share/doc/cvs-version/FAQ**.

CVS information pages are available at <http://ximbiot.com/cvs/manual/>.

The CVS home page is located at <http://www.nongnu.org/cvs/>.

2.2. Apache Subversion (SVN)

Subversion is a version control system that manages files and directories, the changes made to them, and can recover and examine them in case of a fault. It was created to match CVS's features and preserve the same development model, and to address any problems often encountered with CVS. This allowed CVS users to convert to SVN with minimal effort.

This section will cover the installation of SVN and provide details on the everyday uses of SVN.

2.2.1. Installation

SVN can be installed with a binary package, directly from source code, or from the console.

The easiest way to install SVN would be through the console with the command **yum install subversion**. Selecting this option ensures that only Red Hat certified packages are used and removes the requirement to manually update them.

Finally, SVN can be installed from source code, though this can be quite complex. From the SVN website, download the latest released source code and follow the instructions in the **install** file.

2.2.2. SVN Repository

In order to begin using SVN, first create a new repository. SVN has no way to determine the difference between projects; it is up to the user to administer the file tree and place the project in separate directories as they prefer. Use the following commands to create the repository:

```
# mkdir /var/svn
# svnadmin create /var/svn/repos
# ls /var/svn/repos/
conf db format hooks locks README.txt
```

This command will create the new directory **/var/svn/repos** with the required database files.

The SVN repository is accessed with a URL. Usually these use the standard syntax of **http://** but it is not limited by this. It also accepts the following URL forms:

file:///

Direct repository access (on local disk)

http://

Access with WebDAV protocol to Subversion-aware Apache server

https://

Same as http:// but with SSL encryption

svn://

Access via custom protocol to an **svnserver** server

svn+ssh://

Same as svn:// but through an SSH tunnel.

**Important**

If the URL contains spaces place quotation marks around it to ensure the shell treats it as a single argument. Otherwise the URL will be invalid.

2.2.3. Importing Data

Assuming that a project consisting of multiple files has already been created, organize them so that they are all in one directory. It is recommended that you use three top-level directories named **branches**, **tags**, and **trunk**. This is not required by SVN but it is a popular convention. The **trunk** directory should contain the projects files, and the **branches** and **tags** directories should remain empty. For example:

```
myproject/branches/
myproject/tags/
myproject/trunk
  foo.c
  bar.c
  Makefile
```

Once the information has been organized appropriately it is time to import it into the SVN repository. This is done with the **svn import** command. For example:

```
$ svn import /path/to/mytree \
  http://host.example.com/svn/repo/myproject \
  -m "Initial import"
Adding myproject/foo.c
Adding myproject/bar.c
Adding myproject/subdir
Adding myproject/subdir/quux.h

Committed revision 1.
$
```

SVN creates the required directories based on how the file tree is set up. It can now be viewed at the URL created, or by the command:

```
$ svn list http://host.example.com/svn/repo/myproject
```

2.2.4. Working Copies

Now that the first revision of the project has been checked into the repository, it can be edited and worked on. To do this, create a working copy. This is done with the **svn checkout** command. For example:

```
$ svn checkout http://host.example.com/svn/repo/trunk
A trunk/README
A trunk/INSTALL
A trunk/src/main.c
A trunk/src/header.h
...
Checked out revision 8810.
$
```

A directory with a working copy of the project is now created on the local machine. It is also possible to specify where the local directory a project is checked out to with the following command:

```
$ svn checkout http://host.example.com/svn/repo/trunk my-working-copy
```

If the local directory specified does not exist, SVN will create it.



Warning

Every directory in the working copy contains a subdirectory called **.svn**. Being an administrative directory, it will not usually appear with a list command. This is an important file and should not be deleted or changed. Subversion uses this directory to manage the working copy and tampering with it will cause errors and instability. If the directory is accidentally deleted the best way to fix it is to delete the entire containing directory (a normal system delete, not **svn delete**) and run **svn update** from a parent directory. The deleted directory will be recreated, including the missing or changed **.svn** directory. This can cause a loss of data.

Although the working copy is now ready to edit, keep in mind that whenever the file tree changes, these changes must be sent to the repository as well. This is done with a variety of commands.

svn add filename

Newly created files or directories, including the files they contain, are flagged to be added to the repository. The next commit will add them to the repository where they can be accessed and viewed by all.

svn delete filename

Files or directories, including the files they contain, are flagged to be deleted from the repository. The next commit will remove them. However, the deleted files can still be accessed in previous revisions through SVN.

svn copy filename1 filename2

Creates a new file, *filename2*, which is an exact copy of *filename1*. It then schedules *filename2* for addition on the next commit. Note that **svn copy** does not create intermediate directories unless the **--parents** option is passed.

svn move filename1 filename2

This is the same as **svn copy filename1 filename2** followed by **svn delete filename1**. A copy is made, and then *filename1* is scheduled to be deleted on the next commit. Note that **svn move** does not create intermediate directories unless the **--parents** option is passed.

svn mkdir directory

This command both creates the specified directory and then schedules it to be added to the repository on the next commit.

Sometimes it is impractical to check out an entire working copy in order to do some simple changes. In these circumstances it is possible to perform **svn mkdir**, **svn copy**, **svn move**, and **svn delete** directly on the repository URL.



Important

Be careful when using these commands as there is no way to check the results with a working copy first.

2.2.5. Committing Changes

Once the edits are complete and have been verified to work correctly, it is time to publish them so others can view the changes.

For each file in the working copy, SVN records two pieces of information:

- ▶ The file's working revision that the current working file is based on.
- ▶ A timestamp recording when the local copy was last updated by the repository.

Using this information, SVN sorts the working copy on the local system into four categories:

Unchanged; current

The file in the working directory is unchanged and matches the copy in the repository, meaning no changes have been committed since the initial check out. Both **svn commit** and **svn update** will do nothing.

Locally changed; current

The file in the working directory has been edited but has not yet been committed to the repository, and the repository version has not been changed since the initial checkout. Running **svn commit** will update the repository with the changes in the working directory; running **svn update** will do nothing.

Unchanged; out of date

The file in the working directory has not been edited, but the version in the repository has, meaning that the working copy is now out of date. Running **svn commit** will do nothing; running **svn update** will merge the changes in the repository with the local working copy.

Locally changed; out of date

The file in both the working directory and the repository has been changed. If **svn commit** is run first, an 'out-of-date' error will occur. Update the file first. Running **svn update** will attempt to merge the changes in the repository with those on the working copy. If there are conflicts SVN will provide options for the user to decide on the best course of action to resolve them.

Running **svn status** will display all of the files in the working tree that do not match the current version in the repository, coded by a letter.

? item

The file is not recognized by SVN; that is it is in the working copy, but has not yet been added to the repository, or been scheduled for any action.

A item

The file is scheduled for addition to the repository and will be added on the next commit.

C item

The file is in conflict with a change made on the repository. This means that someone has edited and committed a change to the same section of the file currently changed in the working copy, and SVN does not know which is 'correct'. This conflict must be resolved before the changes are committed.

D item

The file is scheduled for deletion on the next commit.

M item

The file has been modified and the changes will be updated on the next commit.

If the **--verbose (-v)** is passed with **svn status**, the status of every item in the working copy will be displayed, even those that have not been changed. For example:

```
$ svn status -v
M 44 23 sally README
 44 30 sally INSTALL
M 44 20 harry bar.c
 44 18 ira stuff
 44 35 harry stuff/trout.c
D 44 19 ira stuff/fish.c
 44 21 sally stuff/things
A 0 ? ? stuff/things/blou.h
 44 36 harry stuff/things/gloo.c
```

Along with the letter codes, this shows the working revision, the revision in which the item was last changed, who changed it, and the item changed respectively.

It can also be useful to see which items have been modified in the repository since the last time a checkout was performed. This is done by passing the **--show-updates (-u)** with **svn status**. An asterisk (*) will be displayed between the letter status and the working revision number on any files that will be updated when performing an **svn commit**.

Another way to view changes made is with the **svn diff** command. This displays changes in a unified diff format, describing changes as 'snippets' of a file's content where each line is prefixed with a character: a space for no change, a minus sign (-) for a line removed, and a plus sign (+) for an added line.

Occasionally a conflict will occur. SVN provides the three most common responses (postpone, diff-full, and edit) and a fourth option to list all the options and what they each do. The options available are:

(p) postpone

Mark the conflict to be resolved later.

(df) diff-full

Display the differences between the base revision and the conflicted file in unified diff format.

(e) edit

Change merged file in an editor.

(r) resolved

Accept the merged version of the file.

(mf) mine-full

Accept my version of the entire file, ignoring the most recent changes in the repository.

(tf) theirs-full

Accept their version of the entire file, ignoring the most recent changes in the local working copy.

(l) launch

Launch an external tool to resolve conflict (this requires set up of the chosen external tool beforehand).

(h) help

Displays the list of options as detailed here.

Finally, provided the project has been changed locally and any conflicts have been resolved, the changes can be successfully committed with the **svn commit** command, appending the option **-m**:

```
$ svn commit filename -m"Fixed a typo in filename"
Sending filename
Transmitting file data .
Committed revision 57.
$
```

As can be seen above, the **-m** option allows a commit message to be recorded. This is most useful when the message is meaningful, which in turn makes referring back over commits straightforward.

The most updated version is now available for anyone with access to the repository to update their versions to the newest copy.

2.2.6. SVN Documentation

The command **svn --help** provides information on the available commands to be used in conjunction with SVN and **svn subcommand --help** provides more detailed information on the specified subcommand.

The official SVN book is available online at <http://svnbook.red-bean.com/>

The official SVN website is located at <http://subversion.apache.org/>

2.3. Git

Git is a version control system that was not written to improve on CVS and SVN but rather in retaliation to them. Git was created with four design points in mind:

1. Not like CVS and SVN. Torvalds, the creator of Git, does not like these programs and wanted to make something that was unlike them.
2. Support a BitKeeper-like workflow. The way a project is managed ideally follows the same process as BitKeeper, while keeping its own design and not becoming a BitKeeper clone.
3. Strong safeguards against accidental or malicious corruption.
4. Very high performance.

To accomplish this, Git approaches how it handles data differently to its predecessors.

This section will go through the most common processes in a day's use of Git.

Previously the version controls covered (CVS and SVN) treated data as changes to a base version of each file. Instead, Git treats its data changes as separate snapshots of what the files look like and stores a reference to that file (though if the file remains unchanged, Git will store a link to the previous identical version rather than copy another file). This creates a kind of new mini file system. The image below compares these concepts visually:

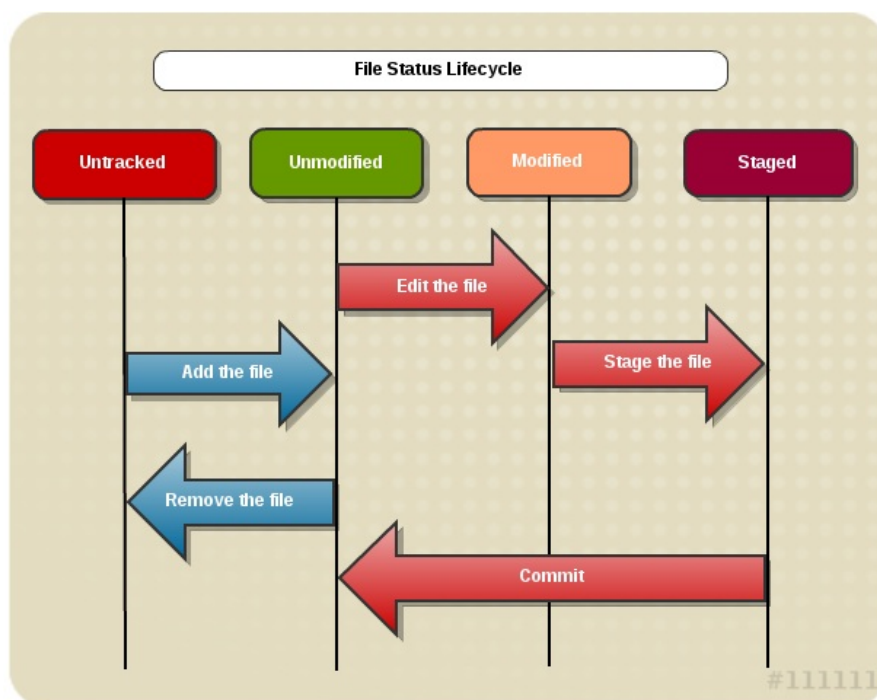


Figure 2.1. Git Version Control

Git is particularly fast, aided by not having to constantly connect to a remote repository. The snapshot nature of Git and how all versions are stored on the local file system means that nearly everything can be done without connecting to any kind of network and the history of the project is available locally.

To fulfill Torvalds' integrity requirement, everything in Git is check-summed before being stored and then referred to by that check-sum. This means the contents cannot be changed without Git's knowledge and information cannot be lost in transit or corrupted. A SHA-1 hash mechanism (a forty-character hexadecimal string) is used for this.

In addition, there is very little in Git that cannot be undone. This is aided by the three main states a file can reside in.

Committed

Data is safely stored on the local database, and unchanged.

Modified

The file has been changed but not yet committed to the database.

Staged

A modified file has been marked to be committed in its current version.

2.3.1. Installation

Git can be installed either from source or from the console. If the user is confident enough then the recommendation is to install from source, as the binary installers do not always have the most up-to-date version available.

To install Git from source code, use the following procedure:

Procedure 2.2. To install Git from source code

1. Install the libraries Git depends on: **curl**, **zlib**, **openssl**, **expat**, and **libiconv**.

```
# yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel gcc
```

2. Download the latest snapshot from the Git web site, located here: <http://git-scm.com/download>.
3. Compile and install.

```
# tar -zxf git-1.7.6.1.tar.gz
# cd git-1.7.6.1
# make prefix=/usr/local
# make prefix=/usr/local install
```

4. It is now possible to get updates for Git, from Git.

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Installing Git with a binary installer from the console is as simple as using the following command.

```
# yum install git
```

2.3.2. Initial Setup

After installing there are a few steps to personalize Git and get it ready for use. Once these settings are configured, they persist for future Git sessions, however to be change them again in the future, run the commands again.

These changes are made by altering variables stored in three different places:

1. The `/etc/gitconfig` file contains variables for every user on the system and all their repositories. It holds the base settings and passing `--system` to `git config` sets it to read and write from this file.
2. The `~/.gitconfig` file is specific to the user. Passing `--global` tells Git to read and write to this file, overriding the settings made in the first point.
3. The config file in the Git directory (`.git/config`) of the repository currently being used. This is specific to this repository only and overrides the settings in both the first and the second point.

Before the first commit, enter some details into Git by supplying the name and email address that will appear with change.

For example, if the user's name is John Q. Smith, use the following commands:

```
$ git config --global user.name "John Smith"
$ git config --global user.email "jsmith@example.com"
```

As explained above, by passing the `--global` option these settings remain constant, but can be overridden for specific repositories.

By default, whenever an editor is required, Git launches Vi or Vim. However, if this is not preferred it is possible to change this to another editor. To do so, use the following command:

```
git config --global core.editor EditorName
```

The diff tool is often used to view the changes in various files, useful for double checking things before committing them.

Use the following command to set the preferred diff tool:

```
$ git config --global merge.tool DiffTool
```

Finally, it is useful to check these settings to ensure they are correct. To do this run:

```
$ git config --list
user.name=John Smith
user.email=jsmith@example.com
```

If there are different settings in different files, Git will list them all, with the last value for the active one. It is also possible for Git to check the specific response to a variable by using the `git config {key}` command. For example:

```
$ git config user.name
John Smith
```

2.3.3. Git Repository

The Git repository is where the metadata and object database is stored for a project. This is where the project is pulled from in order to get a local clone of a repository on a local system.

There are two options for obtaining a Git repository. The first is used when a directory already exists and is required to initialize a Git repository. The second is cloning a repository that already exists.

To clone an existing repository (for example, to contribute to) then run the following command:

```
$ git clone git://location/of/git/repository.git
```

Note that the command is **git clone** as opposed to **git checkout** as it might be for a version control system similar to CVS and SVN. This is because Git receives a copy of every file in the project's entire history, as opposed to only the most recent files as with other version control systems.

The above command creates a directory where the name is the last component of the URL, but with any **.git** suffix removed. However, the **clone** command can use any other name by appending the desired directory name to the end:

```
$ git clone git://location/of/git/repository.git my_git_repo
```

Finally, even though this command uses the **git://** protocol, it is also possible to use **http://** or **https://** as appropriate.

To create a new Git repository ready to create data for, first navigate to the project's directory and type:

```
$ git init
```

This creates a skeleton of a Git repository, containing all the necessary files ready for content to be created and added.

Now that either a skeleton Git repository is set up or a local clone copied and ready on the local system it is time to look at the rest of the Git cycle.

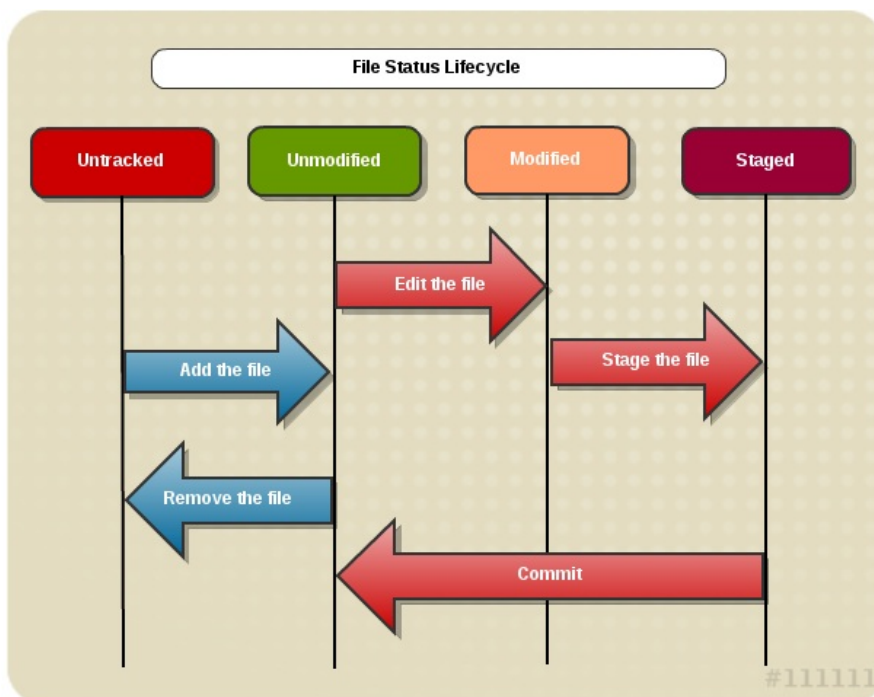


Figure 2.2. Git Life Cycle

This image shows how the Git cycle works and will be explained in further detail in the following sections.

2.3.4. Untracked Files

Untracked files are those that Git does not recognize. This will occur if a file is newly created or for all files in a new project. The status of a file can be shown with the **git status** command. For a newly started project there will be files in the untracked status.

```
$ git status
# On branch master
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#  filename
nothing added to commit but untracked files present (use "git add" to track)
```

As the status helpfully says, the files will not be included unless Git is told to include them with the **git add** command.

```
$ git add filename
```

The command **git add filename** will add that specific file first to the unmodified section. Use **git add .** to add all files in the current directory (including any sub-directories), or for example **git add *.[ch]** to add all **.c** and **.h** files in the current directory.

2.3.5. Unmodified Files

The unmodified status is where those files that have not changed are kept. Git is aware of them and is tracking them so that when an edit is made they are transferred to the modified status. Also, after a commit, the files are returned to this state.

It is also possible to remove files from this state to stop Git from tracking them. This will remove them locally as well. To do so run:

```
$ git rm filename
rm 'filename'
$ git status
# On branch master
#
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#   deleted: filename
#
```



Important

Note, that if a file is unmodified, `git rm filename` will remove the entire file. It is only when a file has uncommitted changes that `git rm filename` will give a diagnostic and not remove it. To remove a file despite the uncommitted changes, use the `--force` or `-f` option.

To stop Git from tracking a file without removing it locally, use the `--cached` option, then commit the removal.

```
$ git rm --cached filename
$ git commit -m'remove file message'
```

2.3.6. Modified Status

A copy is on the local system ready to edit. As soon as any changes are made Git recognizes the file as modified and moves it to the modified status. Running `git status` will show this:

```
$ git status
# On branch master
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
#   modified:   filename
#
```

The file has been changed but as of yet it will not be committed (after all, more changes can still be made and it may not be ready to be committed). As the output helpfully points out, using the `git add filename` command again will push the modified file to the staged status, ready to be committed.

```
$ git add filename
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#   new file:   filename
#
```

This process can become a little more complex if a staged file requires one final edit before it is committed as it will appear in both the staged status and the modified status. If this occurs then a status will look like this:

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#   modified:   filename1
#
# Changed but not updated:
# (use "git add <file>..." to unstage)
#
#   modified:   filename1
#
```

This is where the Git snapshots are highlighted; there is a snapshot of a file ready to be committed and another snapshot in the modified status. If a commit is run then only the snapshot of the staged status will be committed, not the corrected version. Running **git add** again will resolve this and the modified snapshot of the file will merge with the snapshot on the staged status, ready to commit the new changes.

2.3.7. Staged Files

The staged status is where the snapshot of all files that are ready to be committed reside. All files in this status will be committed when the command is given.

2.3.7.1. Viewing changes

Before committing the snapshots on the staged status, it is a good idea to check the changes made to ensure that they are acceptable. This is where the command **git diff** comes in.

```
$ git diff
diff --git a/filename b/filename
index 3cb747f..da65585 100644
--- a/filename
+++ b/filename
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
    end

+ some code
+ some more code
+ a comment
+ another change
- a mistake
```

Running the **git diff** command with no parameters, as above, compares the working directory to what is in the staged status, displaying changes made but not yet committed.

It is also possible to compare the changes between the staged status and what the last commit was by using the **--cached** option.

```
$ git diff --cached
diff --git a/filename b/filename
new file mode 100644
index 0000000..03902a1
-- /dev/null
+++ b/filename
@@ -0,0 +1,5 @@
+file
+ by name1, name2
+ http://path/to/file
+
+ added information to file
```



Note

In versions 1.6.1 and later of Git it is also possible to use the **--staged** option instead of **--cached**.

2.3.7.2. Committing Changes

After checking that all the staged files are correct and ready, it is time to commit the changes.

```
$ git commit
```

The above command will launch the chosen editor set in the initial setup, or if this was not set up it defaults to Vim.

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#   new file:   filename2
#   modified:   filename1
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

As can be seen, the last status report is also included in the editor, though because of the hash (#) it will not be visible in the actual commit log. For more information on what is being committed, pass the **-v** option with **git commit**. The commit message can be entered here, or it can be entered in the command line with the **-m** option:

```
$ git commit -m "commit message"
[master]: created 4156dc4f: "commit message"
2 files changed, 3 insertions(+), 1 deletions (-)
create mode 100644 filename
```

The commit message provides some information: the branch committed to (master in this case), what SHA-1 checksum the commit has (4156dc4f), how many files were changed, and statistics about what was changed within them.



Note

It is possible to skip the staging area which can be useful, but holds the risk of committing something that was not ready to be committed yet. This is done by passing the **-a** option with **git commit**.

```
$ git status
# On branch master
#
# Changed but not updated:
#
#   modified:   filename
#
$ git commit -a -m 'commit message'
[master 16e15c7] commit message
1 files changed, 5 insertions(+), 2 deletions(-)
```

2.3.8. Remote Repositories

In order to share a project with the world, push it to a remote repository, hosted on a network or on the internet. To be able to push to a remote directory, first one must be made. To do so, run the command **git add [shortname] [URL]**

```
$ git remote add shortname git://path/to/new/repo
```

The shortname can now be used in lieu of the URL when referring to the remote repository.

Now a repository is added it is possible to print a list of remote repositories. This is done with the **git remote** command, passing the **-v** option to display the associated URL as well as just the shortname if desired. Even without adding a new remote repository, if the project was cloned it will list at least the repository it was cloned from.

```
$ git remote -v
repo-name git://path/to/new/remote-repository
origin git://path/to/original/remote-repository
```

If even this is not enough information running the **git show [remote-repository]** will list the URL as well as the branches Git is tracking for this particular repository.

In order to fetch data from remote repositories (for example, if someone working on the project has pushed new information to them), use the following command:

```
$ git fetch [remote-name]
```

This pulls down any and all information from this particular remote host that differs from what is on the local copy. Alternatively, running **git pull** will do the same from the repository the original copy was cloned from.

In order to share a project with a wider audience it is required to push it to a remote repository.

```
$ git push remote-repository branch-name
```

This command pushes the specified branch to the specified repository, but only if the user has write access. In the case of a conflict, pull the new work down first to incorporate it into the local version before pushing it to the remote repository.

Finally to rename a repository, run the **git remote rename original-name new-name**. Keep in mind that this will also change the remote branch names as well. Removing a repository is similar: **git remote rm remote-name**.

2.3.9. Commit Logs

After working with a repository for some time, making several changes and commits, it may be required to view the logs of these changes. This is done with the **git log** command. When this is run with no arguments, it lists each commit in reverse chronological order, presenting the time and date of the commit, the SHA-1 checksum, the author's name and email, and the commit message.

It is possible to include the diff report in these logs, and limit the output by a set number of entries. For example, running **git log -p -2** will list the normal log information in addition to the diff reports for the two most recent entries.

To include some statistics at the end of each commit entry, use the **git log --stat**. This command will include a list of modified files, how many files were changed, and how many lines were added and removed, followed by a summary of this information, after the commit message.

Along with these useful options, there are a number of others which can be passed with this command:

--shortstat

Similar to the **--stat** option, but this displays only the changed, insertions, and deletions in a commit.

--name-only

Lists only the files modified after the commit information.

--name-status

Lists the files modified along with the changed, insertions, and deletions.

--abbrev-commit

Only displays the first few characters of the SHA-1 checksum, instead of all forty.

--relative-date

Displays the date relative to the current date. For example, instead of reading **Tue July 10:53:11 2011 -0700**, it will print **2 weeks ago**.

--graph Display

Prints an ASCII graph of the branch and merge history beside the log output.

--since=[date]

Prints the log since a specified date; that is, everything after the date. The date can be entered in a variety of different formats, be it a specific date (2010-08-23, for example) or a relative date ("1 year 3 days 4 minutes ago", for example).

--until=[date]

Similar to the **--since** option, this will print the log up until the specified date; that is, everything before the date.

--author name

Lists only those commits with the specified name in the author field.

--committer name

Lists only those commits with the specified name in the committer field.

2.3.10. Fixing Problems

There may come a time when mistakes are made and something has to be removed or undone. This section will cover some of the ways these errors can be fixed.



Warning

This is one of the few areas in Git where data can be lost forever. If an undo is performed and then discovered it should not have been, it is highly likely that it is now impossible to recover the lost data. Proceed with caution.

This occurs most often when a commit is pushed too early, committing something that is not yet ready, or making a mistake in the commit message. This can be fixed by committing over the top of the latest commit using the **--amend** option.

```
$ git commit --amend
```

If the files on the staged status are different from those in the latest commit, the commit will run normally except it will override the original. If the files are the same, then the option will be provided to change the commit message, again, overriding the previous commit.

It is also possible to unstage a staged file. This can sometimes be required if **git add *** was used when the intention was to have two (or more) separate commits. The **git status** command provides a hint on how to do this as well:

```
# (use "git reset HEAD <file>..." to unstage)
```

So to follow its advice, use the command **git reset HEAD filename** and the file is now reverted to the modified status rather than the staged status.

To revert a file back to what it looked like at the last commit, the **git status** command comes to the rescue again in the unstaged status:

```
# (use "git checkout -- <file>..." to discard changes in working directory)
```

Following these instructions with the **git checkout -- filename** reverts the file. To reiterate the above warning however, this *will cause data loss*; only use it when it is certain this version of the file is no longer wanted.

2.3.11. Git Documentation

The main Git man page can be viewed with **man git**. This also provides the commands to access other man pages such as **gittutorial(7)**, **Everyday Git[1]**, and **gitglossary(7)**.

The official Git homepage can be accessed at <http://git-scm.com/> where more documentation is available and can be downloaded.

The following is a list of websites containing more detailed Git information.

- ▶ A useful book containing both basic and more advanced Git instructions can be found at <http://progit.org/book/>.
- ▶ A Red Hat Magazine article, *Shipping quality code with Git*: <http://magazine.redhat.com/2008/05/02/shipping-quality-code-with-git/>.
- ▶ *Using Git without feeling stupid - part 1*: <http://smalltalk.gnu.org/blog/bonzinip/using-git-without-feeling-stupid-part-1>.

Using Git without feeling stupid - part 2: <http://smalltalk.gnu.org/blog/bonzinip/using-git-without-feeling-stupid-part-2>.

- » A description on how to prepare or email patches: <http://logcheck.org/git.html>.
- » A Git cheat sheet: <http://cheat.errtheblog.com/s/git>.
- » A blog by Tim Waugh, *How I use Git*: <http://cyberelk.net/tim/2009/02/04/how-i-use-git/>.
- » *Handling and avoiding conflicts in Git*: <http://weblog.masukomi.org/2008/07/12/handling-and-avoiding-conflicts-in-git>.
- » *Branching and merging with Git*: <http://lwn.net/Articles/210045/>.
- » *Getting the most out of Git in GNOME*: <http://live.gnome.org/Git/Developers>.
- » *Integrating vim with Git*: <http://vim.runpaint.org/extending/integrating-vim-with-git/>.
- » *Git Reference*: <http://gitref.org/>.
- » *A successful Git branching model*: <http://nvie.com/posts/a-successful-git-branching-model/>.
- » A quick introduction, *Git for the lazy*: http://www.spheredev.org/wiki/Git_for_the_lazy.
- » *Git tricks, tips, and workflows*: <http://nuclearsquid.com/writings/git-tricks-tips-workflows/>.

Chapter 3. Libraries and Runtime Support

Red Hat Enterprise Linux supports the development of custom applications in a wide variety of programming languages using proven, industrial-strength tools. This chapter describes the runtime support libraries provided in Red Hat Enterprise Linux 7.

3.1. Version Information

The following table compares the version information for runtime support packages in supported programming languages between Red Hat Enterprise Linux 7, Red Hat Enterprise Linux 6, Red Hat Enterprise Linux 5, and Red Hat Enterprise Linux 4.

This is not an exhaustive list. Instead, this is a survey of standard language runtimes, and key dependencies for software developed on Red Hat Enterprise Linux 7.

Table 3.1. Language and Runtime Library Versions

Package Name	Red Hat Enterprise 7	Red Hat Enterprise 6	Red Hat Enterprise 5	Red Hat Enterprise 4
<i>glibc</i>	2.12	2.12	2.5	2.3
<i>libstdc++</i>	4.8	4.4	4.1	3.4
<i>boost</i>	1.53	1.41	1.33	1.32
<i>java</i>	1.7	1.5 (IBM), 1.6 (IBM, OpenJDK, Oracle Java)	1.4, 1.5, and 1.6	1.4
<i>python</i>	2.7	2.6	2.4	2.3
<i>php</i>	5.4	5.3	5.1	4.3
<i>ruby</i>	2.0	1.8	1.8	1.8
<i>httpd</i>	2.4	2.2	2.2	2.0
<i>postgresql</i>	9.2	8.4	8.1	7.4
<i>mysql</i>	5.4	5.1	5.0	4.1
<i>nss</i>	3.15	3.12	3.12	3.12
<i>openssl</i>	1.0.1e	1.0.0	0.9.8e	0.9.7a
<i>libX11</i>	1.6	1.3	1.0	
<i>firefox</i>	24.4	3.6	3.6	3.6
<i>kdebase</i>	4.10	4.3	3.5	3.3
<i>gtk2</i>	2.24	2.18	2.10	2.04

Note

The **compat-glibc** RPM is included with Red Hat Enterprise Linux 7, but it is not a runtime package and therefore not required for running anything. It is solely a development package, containing header files and dummy libraries for linking. This allows compiling and linking packages to run in older Red Hat Enterprise Linux versions (using **compat-gcc-*** against those headers and libraries). Running **rpm -qpi compat-glibc-*** will provide some information on how to use this package.

For more information on **compat-glib**, see [Section 3.3.1, “compat-glibc”](#)

3.2. Compatibility

Compatibility specifies the portability of binary objects and source code across different instances of a computer operating environment. Officially, Red Hat supports current release and two consecutive prior versions. This means that applications built on Red Hat Enterprise Linux 4 and Red Hat Enterprise Linux 5 will continue to run on Red Hat Enterprise Linux 6 as long as they comply with Red Hat guidelines (using the symbols that have been white-listed, for example).

Red Hat understands that as an enterprise platform, customers rely on long-term deployment of their applications. For this reason, applications built against C/C++ libraries with the help of compatibility libraries continue to be supported for ten years.

There are two types of compatibility:

Source Compatibility

Source compatibility specifies that code will compile and execute in a consistent and predictable way across different instances of the operating environment. This type of compatibility is defined by conformance with specified Application Programming Interfaces (APIs).

Binary Compatibility

Binary Compatibility specifies that compiled binaries in the form of executables and *Dynamic Shared Objects* (DSOs) will run correctly across different instances of the operating environment. This type of compatibility is defined by conformance with specified Application Binary Interfaces (ABIs).

For further information regarding this and all levels of compatibility between core and non-core libraries, refer to Red Hat Enterprise Linux supported releases accessed at <https://access.redhat.com/support/policy/updates/errata/> and the general Red Hat Enterprise Linux compatibility policy, accessed at <https://access.redhat.com/site/articles/119073>.

3.2.1. Static Linking

Static linking is emphatically discouraged for all Red Hat Enterprise Linux releases. Static linking causes far more problems than it solves, and should be avoided at all costs.

The main drawback of static linking is that it is only guaranteed to work on the system on which it was built, and even then only until the next release of glibc or libstdc++ (in the case of C++). There is no forward or backward compatibility with a static build. Furthermore, any security fixes (or general-purpose fixes) in subsequent updates to the libraries will not be available unless the affected statically linked executables are re-linked.

A few more reasons why static linking should be avoided are:

- ▶ Larger memory footprint.
- ▶ Slower application startup time.
- ▶ Reduced glibc features with static linking.
- ▶ Security measures like load address randomization cannot be used.
- ▶ Dynamic loading of shared objects outside of glibc is not supported.

For additional reasons to avoid static linking, see: [Static Linking Considered Harmful](#).

3.3. Library and Runtime Details

3.3.1. `compat-glibc`

`compat-glibc` provides a subset of the shared static libraries from previous versions of Red Hat Enterprise Linux. For Red Hat Enterprise Linux 6, the following libraries are provided:

- ▶ `libanl`
- ▶ `libcidn`
- ▶ `libcrypt`
- ▶ `libc`
- ▶ `libdl`
- ▶ `libm`
- ▶ `libnsl`
- ▶ `libpthread`
- ▶ `libresolv`
- ▶ `librt`
- ▶ `libthread_db`
- ▶ `libutil`

This set of libraries allows developers to create a Red Hat Enterprise Linux 5 application with Red Hat Enterprise Linux 6, provided the application uses only the above libraries. Use the following command to do so:

```
# gcc -fgnu89-inline -I /usr/lib/x86_64-redhat-linux5E/include -B  
/usr/lib/x86_64-redhat-linux5E/lib64/ -lc_nonshared
```



Important

Applications that violate the ISO with regards to overlapping source or destination memory locations for `memcpy` and other functions will likely fail.

3.3.2. The GNU C++ Standard Library

The `libstdc++` package contains the GNU C++ Standard Library, which is an ongoing project to implement the ISO 14882 Standard C++ library.

Installing the `libstdc++` package will provide just enough to satisfy link dependencies (that is, only shared library files). To make full use of all available libraries and header files for C++ development, you must install `libstdc++-devel` as well. The `libstdc++-devel` package also contains a GNU-specific implementation of the Standard Template Library (STL).

For Red Hat Enterprise Linux 4, 5, and 6, the C++ language and runtime implementation has remained

stable and as such no compatibility libraries are required for **libstdc++**. However, this is not the case for Red Hat Enterprise Linux 2 and 3. For Red Hat Enterprise Linux 2 **compat-libstdc++-296** is required to be installed. For Red Hat Enterprise Linux 3 **compat-libstdc++-33** is required to be installed. Neither of these are installed by default so have to be added separately.

3.3.2.1. GNU C++ Standard Library Updates

The Red Hat Enterprise Linux 6 version of the GNU C++ Standard Library features the following improvements over its Red Hat Enterprise Linux 5 version:

- » Added support for elements of ISO C++ TR1, namely:
 - `<tr1/array>`
 - `<tr1/complex>`
 - `<tr1/memory>`
 - `<tr1/functional>`
 - `<tr1/random>`
 - `<tr1/regex>`
 - `<tr1/tuple>`
 - `<tr1/type_traits>`
 - `<tr1/unordered_map>`
 - `<tr1/unordered_set>`
 - `<tr1/utility>`
 - `<tr1/cmath>`
- » Added support for elements of the upcoming ISO C++ standard, C++0x. These elements include:
 - `<array>`
 - `<chrono>`
 - `<condition_variable>`
 - `<forward_list>`
 - `<functional>`
 - `<initializer_list>`
 - `<mutex>`
 - `<random>`
 - `<ratio>`
 - `<regex>`

- `<system_error>`
 - `<thread>`
 - `<tuple>`
 - `<type_traits>`
 - `<unordered_map>`
 - `<unordered_set>`
- ▶ Added support for the `-fvisibility` command.
 - ▶ Added the following extensions:
 - `__gnu_cxx::typelist`
 - `__gnu_cxx::throw_allocator`

For more information about updates to `libstdc++` in Red Hat Enterprise Linux, refer to the *C++ Runtime Library* section of the following documents:

- ▶ *GCC 4.2 Release Series Changes, New Features, and Fixes*: <http://gcc.gnu.org/gcc-4.2/changes.html>
- ▶ *GCC 4.3 Release Series Changes, New Features, and Fixes*: <http://gcc.gnu.org/gcc-4.3/changes.html>
- ▶ *GCC 4.4 Release Series Changes, New Features, and Fixes*: <http://gcc.gnu.org/gcc-4.4/changes.html>

3.3.2.2. GNU C++ Standard Library Documentation

To use the `man` pages for library components, install the `libstdc++-docs` package. This will provide `man` page information for nearly all resources provided by the library; for example, to view information about the `vector` container, use its fully-qualified component name:

```
man std::vector
```

This will display the following information (abbreviated):

```
std::vector(3)                                std::vector(3)
NAME
  std::vector -
  A standard container which offers fixed time access to individual
  elements in any order.
SYNOPSIS
  Inherits std::_Vector_base< _Tp, _Alloc >.
  Public Types
  typedef _Alloc allocator_type
  typedef __gnu_cxx::__normal_iterator< const_pointer, vector >
    const_iterator
  typedef _Tp_alloc_type::const_pointer const_pointer
  typedef _Tp_alloc_type::const_reference const_reference
  typedef std::reverse_iterator< const_iterator >
```

The **libstdc++-docs** package also provides manuals and reference information in HTML form at the following directory:

file:///usr/share/doc/libstdc++-docs-version/html/spine.html

The main site for the development of libstdc++ is hosted on gcc.gnu.org.

3.3.3. Boost

The **boost** package contains a large number of free peer-reviewed portable C++ source libraries. These libraries are suitable for tasks such as portable file-systems and time/date abstraction, serialization, unit testing, thread creation and multi-process synchronization, parsing, graphing, regular expression manipulation, and many others.

Installing the **boost** package will provide just enough libraries to satisfy link dependencies (that is, only shared library files). To make full use of all available libraries and header files for C++ development, you must install **boost-devel** as well.

The **boost** package is actually a meta-package, containing many library sub-packages. These sub-packages can also be installed individually to provide finer inter-package dependency tracking. The meta-package includes all of the following sub-packages:

- ▶ **boost-date-time**
- ▶ **boost-filesystem**
- ▶ **boost-graph**
- ▶ **boost-iostreams**
- ▶ **boost-math**
- ▶ **boost-program-options**
- ▶ **boost-python**
- ▶ **boost-regex**
- ▶ **boost-serialization**
- ▶ **boost-signals**
- ▶ **boost-system**
- ▶ **boost-test**
- ▶ **boost-thread**
- ▶ **boost-wave**

Not included in the meta-package are packages for static linking or packages that depend on the underlying Message Passing Interface (MPI) support.

MPI support is provided in two forms: one for the default Open MPI implementation ^[1], and another for the alternate MPICH2 implementation. The selection of the underlying MPI library in use is up to the user and depends on specific hardware details and user preferences. Please note that these packages can be installed in parallel, as installed files have unique directory locations.

For Open MPI:

- ▶ **boost-openmpi**
- ▶ **boost-openmpi-devel**
- ▶ **boost-graph-openmpi**
- ▶ **boost-openmpi-python**

For MPICH2:

- ▶ **boost-mpich2**
- ▶ **boost-mpich2-devel**
- ▶ **boost-graph-mpich2**
- ▶ **boost-mpich2-python**

If static linkage cannot be avoided, the **boost-static** package will install the necessary static libraries. Both thread-enabled and single-threaded libraries are provided.

3.3.3.1. Boost Updates

The Red Hat Enterprise Linux 6 version of Boost features many packaging improvements and new features.

Several aspects of the **boost** package have changed. As noted above, the monolithic **boost** package has been augmented by smaller, more discrete sub-packages. This allows for more control of dependencies by users, and for smaller binary packages when packaging a custom application that uses Boost.

In addition, both single-threaded and multi-threaded versions of all libraries are packaged. The multi-threaded versions include the **mt** suffix, as per the usual Boost convention.

Boost also features the following new libraries:

- ▶ Foreach
- ▶ Statechart
- ▶ TR1
- ▶ Typeof
- ▶ Xpressive
- ▶ Asio
- ▶ Bitmap
- ▶ Circular Buffer
- ▶ Function Types
- ▶ Fusion
- ▶ GIL

- ▶ Interprocess
- ▶ Intrusive
- ▶ Math/Special Functions
- ▶ Math/Statistical Distributions
- ▶ MPI
- ▶ System
- ▶ Accumulators
- ▶ Exception
- ▶ Units
- ▶ Unordered
- ▶ Proto
- ▶ Flyweight
- ▶ Scope Exit
- ▶ Swap
- ▶ Signals2
- ▶ Property Tree

Many of the existing libraries have been improved, bug-fixed, and otherwise enhanced.

3.3.3.2. Boost Documentation

The **boost-doc** package provides manuals and reference information in HTML form located in the following directory:

```
file:///usr/share/doc/boost-doc-version/index.html
```

The main site for the development of Boost is hosted on boost.org.

3.3.4. Qt

The **qt** package provides the Qt (pronounced "cute") cross-platform application development framework used in the development of GUI programs. Aside from being a popular "widget toolkit", Qt is also used for developing non-GUI programs such as console tools and servers. Qt was used in the development of notable projects such as Google Earth, KDE, Opera, OPIE, VoxOx, Skype, VLC media player and VirtualBox. It is produced by Nokia's Qt Development Frameworks division, which came into being after Nokia's acquisition of the Norwegian company Trolltech, the original producer of Qt, on June 17, 2008.

Qt uses standard C++ but makes extensive use of a special pre-processor called the *Meta Object Compiler* (MOC) to enrich the language. Qt can also be used in other programming languages via language bindings. It runs on all major platforms and has extensive internationalization support. Non-GUI Qt features include SQL database access, XML parsing, thread management, network support, and a unified cross-platform API for file handling.

Distributed under the terms of the GNU Lesser General Public License (among others), Qt is free and open source software. The Red Hat Enterprise Linux 6 version of Qt supports a wide range of compilers, including the GCC C++ compiler and the Visual Studio suite.

3.3.4.1. Qt Updates

Some of the improvements the Red Hat Enterprise Linux 6 version of Qt include:

- ▶ Advanced user experience
 - **Advanced Graphics Effects:** options for opacity, drop-shadows, blur, colorization, and other similar effects
 - **Animation and State Machine:** create simple or complex animations without the hassle of managing complex code
 - Gesture and multi-touch support
- ▶ Support for new platforms
 - Windows 7, Mac OSX 10.6, and other desktop platforms are now supported
 - Added support for mobile development; Qt is optimized for the upcoming Maemo 6 platform, and will soon be ported to Maemo 5. In addition, Qt now supports the Symbian platform, with integration for the S60 framework.
 - Added support for Real-Time Operating Systems such as QNX and VxWorks
- ▶ Improved performance, featuring added support for hardware-accelerated rendering (along with other rendering updates)
- ▶ Updated cross-platform IDE

For more details on updates to Qt included in Red Hat Enterprise Linux 6, refer to the following links:

- ▶ <http://doc.qt.nokia.com/4.6/qt4-6-intro.html>
- ▶ <http://doc.qt.nokia.com/4.6/qt4-intro.html>

3.3.4.2. Qt Creator

Qt Creator is a cross-platform IDE tailored to the requirements of Qt developers. It includes the following graphical tools:

- ▶ An advanced C++ code editor
- ▶ Integrated GUI layout and forms designer
- ▶ Project and build management tools
- ▶ Integrated, context-sensitive help system
- ▶ Visual debugger
- ▶ Rapid code navigation tools

For more information about **Qt Creator**, refer to the following link:

<http://qt.nokia.com/products/appdev/developer-tools/developer-tools#qt-tools-at-a>

3.3.4.3. Qt Library Documentation

The **qt-doc** package provides HTML manuals and references located in `/usr/share/doc/qt4/html/`. This package also provides the *Qt Reference Documentation*, which is an excellent starting point for development within the Qt framework.

You can also install further demos and examples from **qt-demos** and **qt-examples**. To get an overview of the capabilities of the Qt framework, refer to `/usr/bin/qtdemo-qt4` (provided by **qt-demos**).

For more information on the development of Qt, refer to the following online resources:

- ▶ *Qt Developer Blogs*: <http://labs.trolltech.com/blogs/>
- ▶ *Qt Developer Zone*: <http://qt.nokia.com/developer/developer-zone>
- ▶ *Qt Mailing List*: <http://lists.qt.nokia.com/>

3.3.5. KDE Development Framework

The **kde1ibs-devel** package provides the KDE libraries, which build on Qt to provide a framework for making application development easier. The KDE development framework also helps provide consistency across the KDE desktop environment.

3.3.5.1. KDE4 Architecture

The KDE development framework's architecture in Red Hat Enterprise Linux uses KDE4, which is built on the following technologies:

Plasma

Plasma replaces KDesktop in KDE4. Its implementation is based on the **Qt Graphics View Framework**, which was introduced in Qt 4.2. For more information about **Plasma**, refer to <http://techbase.kde.org/Development/Architecture/KDE4/Plasma>.

Sonnet

Sonnet is a multilingual spell-checking application that supports automatic language detection, primary/backup dictionaries, and other useful features. It replaces **kspe112** in KDE4.

KIO

The KIO library provides a framework for network-transparent file handling, allowing users to easily access files through network-transparent protocols. It also helps provides standard file dialogs.

KJS/KHTML

KJS and KHTML are fully-fledged JavaScript and HTML engines used by different applications native to KDE4 (such as **konqueror**).

Solid

Solid is a hardware and network awareness framework that allows you to develop applications with hardware interaction features. Its comprehensive API provides the necessary abstraction to support cross-platform application development. For more information, refer to <http://techbase.kde.org/Development/Architecture/KDE4/Solid>.

Phonon

Phonon is a multimedia framework that helps you develop applications with multimedia functionalities. It facilitates the usage of media capabilities within KDE. For more information, refer to <http://techbase.kde.org/Development/Architecture/KDE4/Phonon>.

Telepathy

Telepathy provides a real-time communication and collaboration framework within KDE4. Its primary function is to tighten integration between different components within KDE. For a brief overview on the project, refer to http://community.kde.org/Real-Time_Communication_and_Collaboration.

Akonadi

Akonadi provides a framework for centralizing storage of *Parallel Infrastructure Management* (PIM) components. For more information, refer to <http://techbase.kde.org/Development/Architecture/KDE4/Akonadi>.

Online Help within KDE4

KDE4 also features an easy-to-use Qt-based framework for adding online help capabilities to applications. Such capabilities include tooltips, hover-help information, and **khelphcenter** manuals. For a brief overview on online help within KDE4, refer to http://techbase.kde.org/Development/Architecture/KDE4/Providing_Online_Help.

KXMLGUI

KXMLGUI is a framework for designing user interfaces using XML. This framework allows you to design UI elements based on "actions" (defined by the developer) without having to revise source code. For more information, refer to <http://developer.kde.org/documentation/library/kdeqt/kde3arch/xmlgui.html>.

Strigi

Strigi is a desktop search daemon compatible with many desktop environments and operating systems. It uses its own **jstream** system which allows for deep indexing of files. For more information on the development of **Strigi**, refer to <http://www.vandenoever.info/software/strigi/>.

KNewStuff2

KNewStuff2 is a collaborative data sharing library used by many KDE4 applications. For more information, refer to <http://techbase.kde.org/Projects/KNS2>.

3.3.5.2. kdelibs Documentation

The **kdelibs-apidocs** package provides HTML documentation for the KDE development framework in `/usr/share/doc/HTML/en/kdelibs4-apidocs/`. The following links also provide details on KDE-related programming tasks:

- ▶ <http://techbase.kde.org/>
- ▶ <http://techbase.kde.org/Development/Tutorials>
- ▶ <http://techbase.kde.org/Development/FAQs>
- ▶ <http://api.kde.org>

3.3.6. GNOME Power Manager

The backend program of the GNOME power management infrastructure is **gnome-power-manager**. It was introduced in Red Hat Enterprise Linux 5 and provides a complete and integrated solution to power management under the GNOME desktop environment. In Red Hat Enterprise Linux 6, the storage-handling parts of **hal** was replaced by **udisks**, and the **libgnomeprint** stack was replaced by print support in **gtk2**.

3.3.6.1. GNOME Power Management Version Guide

This section will detail what versions of **gnome-power-management** are shipped with the various Red Hat Enterprise Linux versions.

In general, however, Red Hat Enterprise Linux 4 ships with GNOME 2.8, Red Hat Enterprise Linux 5 ships with GNOME 2.16, and Red Hat Enterprise Linux 6 ships with GNOME 2.28.

Table 3.2. Desktop Components Comparison

GNOME Power Management Desktop Components	Red Hat Enterprise Linux Version		
	4	5	6
<i>hal</i>	0.4.2	0.5.8	0.5.14
<i>udisks</i>	N/A	N/A	1.0.1
<i>glib2</i>	2.4.7	2.12.3	2.22.5
<i>gtk2</i>	2.4.13	2.10.4	2.18.9
<i>gnome-vfs2</i>	2.8.2	2.16.2	2.24.2
<i>libglade2</i>	2.4.0	2.6.0	2.6.4
<i>libgnomecanvas</i>	2.8.0	2.14.0	2.26.0
<i>gnome-desktop</i>	2.8.0	2.16.0	2.28.2
<i>gnome-media</i>	2.8.0	2.16.1	2.29.91
<i>gnome-python2</i>	2.6.0	2.16.0	2.28.0
<i>libgnome</i>	2.8.0	2.16.0	2.28.0
<i>libgnomeui</i>	2.8.0	2.16.0	2.24.1
<i>libgnomeprint22</i>	2.8.0	2.12.1	N/A
<i>libgnomeprintui22</i>	2.8.0	2.12.1	N/A
<i>gnome-session</i>	2.8.0	2.16.0	2.28.0
<i>gnome-power-manager</i>	N/A	2.16.0	2.28.3
<i>gnome-applets</i>	2.8.0	2.16.0	2.28.0
<i>gnome-panel</i>	2.8.1	2.16.1	2.30.2

3.3.6.2. API Changes for *glib*

There are a number of API changes for *glib* between versions.

Version 2.4 to Version 2.12

Some of the differences in *glib* between version 2.4 and 2.12 (or between Red Hat Enterprise Linux 4 and Red Hat Enterprise Linux 5) are:

- GOption (a command line option parser)

- ▶ GKeyFile (a key/ini file parser)
- ▶ GObject toggle references
- ▶ GMappedFile (a map wrapper)
- ▶ GSlice (a fast memory allocator)
- ▶ GBookmarkFile (a bookmark file parser)
- ▶ Base64 encoding support
- ▶ Native atomic ops on s390
- ▶ Updated Unicode support to 5
- ▶ Atomic reference counting for GObject

Version 2.12 to Version 2.22

Some of the differences in *glib* between version 2.12 and 2.22 (or between Red Hat Enterprise Linux 5 and Red Hat Enterprise Linux 6) are:

- ▶ GSequence (a list data structure that is implemented as a balanced tree)
- ▶ GRegex (a PCRE regex wrapper)
- ▶ Support for monotonic clocks
- ▶ XDG user dirs support
- ▶ GIO (a VFS library to replace gnome-vfs)
- ▶ GChecksum (support for hash algorithms such as MD5 and SHA-256)
- ▶ GTest (a test framework)
- ▶ Support for sockets and network IO in GIO
- ▶ GHashTable performance improvements
- ▶ GMarkup performance improvements

Documentation for *glib*, including indexes of new and deprecated APIs, is shipped in the *glib2-devel* package.

3.3.6.3. API Changes for GTK+

There are a number of API changes for *GTK+* between versions.

Version 2.4 to Version 2.10

Some of the differences in *GTK+* between version 2.4 and 2.10 (or between Red Hat Enterprise Linux 4 and Red Hat Enterprise Linux 5) are:

- ▶ GtkIconView
- ▶ GtkAboutDialog
- ▶ GtkCellView

- ▶ GtkFileChooserButton
- ▶ GtkMenuToolButton
- ▶ GtkAssistant
- ▶ GtkLinkButton
- ▶ GtkRecentChooser
- ▶ GtkCellRendererCombo
- ▶ GtkCellRendererProgress
- ▶ GtkCellRendererAccel
- ▶ GtkCellRendererSpin
- ▶ GtkStatusIcon
- ▶ Printing Support
- ▶ Notebook tab DND support
- ▶ Ellipsis support in labels, progressbars and treeviews
- ▶ Support rotated text
- ▶ Improved themability

Version 2.10 to Version 2.18

Some of the differences in *GTK+* between version 2.10 and 2.18 (or between Red Hat Enterprise Linux 4 and Red Hat Enterprise Linux 5) are:

- ▶ GtkScaleButton
- ▶ GtkVolumeButton
- ▶ GtkInfoBar
- ▶ GtkBuilder to replace libglade
- ▶ New tooltips API
- ▶ GtkMountOperation
- ▶ `gtk_show_uri`
- ▶ Scale marks
- ▶ Links in labels
- ▶ Support runtime font configuration changes
- ▶ Use GIO

Documentation for *GTK+*, including indexes of new and deprecated APIs, is shipped in the *gtk2-devel* package.

3.3.7. NSS Shared Databases

The NSS shared database format, introduced on NSS 3.12, is now available in Red Hat Enterprise 6. This encompasses a number of new features and components to improve access and usability.

Included, is the NSS certificate and key database which are now sqlite-based and allow for concurrent access. The legacy **key3.db** and **cert8.db** are also replaced with new SQL databases called **key4.db** and **cert9.db**. These new databases will store PKCS #11 token objects, which are the same as what is currently stored in **cert8.db** and **key3.db**.

Having support for shared databases enables a system-wide NSS database. It resides in `/etc/pki/nssdb` where globally trusted CA certificates become accessible to all applications. The command `rv = NSS_InitReadWrite("sql:/etc/pki/nssdb");` initializes NSS for applications. If the application is run with root privileges, then the system-wide database is available on a read and write basis. However, if it is run with normal user privileges it becomes read only.

Additionally, a PEM PKCS #11 module for NSS allows applications to load into memory certificates and keys stored in PEM-formatted files (for example, those produced by openssl).

3.3.7.1. Backwards Compatibility

The binary compatibility guarantees made by NSS upstream are preserved in NSS for Red Hat Enterprise Linux 6. This guarantee states that the NSS 3.12 is backwards compatible with all older NSS 3.x shared libraries. Therefore, a program linked with an older NSS 3.x shared library will work without recompiling or relinking, and any applications that restrict the use of NSS APIs to the NSS Public Functions remain compatible with future versions of the NSS shared libraries.

Red Hat Enterprise Linux 5 and 4 run on the same version of NSS as Red Hat Enterprise Linux 6 so there are no ABI or API changes. However, there are still differences as NSS's internal cryptographic module in Red Hat Enterprise Linux 6 is the one from NSS 3.12, whereas Red Hat Enterprise Linux 5 and 4 still use the older one from NSS 3.15. This means that new functionality that had been introduced with NSS 3.12, such as the shared database, is now available with Red Hat Enterprise Linux 6's version of NSS.

3.3.7.2. NSS Shared Databases Documentation

Mozilla's wiki page explains the system-wide database rationale in great detail and can be accessed [here](#).

3.3.8. Python

The **python** package adds support for the Python programming language. This package provides the object and cached bytecode files required to enable runtime support for basic Python programs. It also contains the **python** interpreter and the **pydoc** documentation tool. The **python-devel** package contains the libraries and header files required for developing Python extensions.

Red Hat Enterprise Linux also ships with numerous **python**-related packages. By convention, the names of these packages have a **python** prefix or suffix. Such packages are either library extensions or python bindings to an existing library. For instance, **dbus-python** is a Python language binding for D-Bus.

Note that both cached bytecode (`*.pyc`/`*.pyo` files) and compiled extension modules (`*.so` files) are incompatible between Python 2.4 (used in Red Hat Enterprise Linux 5), Python 2.6 (used in Red Hat Enterprise Linux 6), and Python 2.7 (used in Red Hat Enterprise Linux 7). As such, you will be required to rebuild any extension modules you use that are not part of Red Hat Enterprise Linux.

3.3.8.1. Python Updates

Red Hat Enterprise Linux 7 ships with Python 2.7. For information about these changes, refer to the following project resource:

- ▶ What's New in Python 2.7: <http://docs.python.org/dev/whatsnew/2.7.html>

Both resources also contain advice on porting code developed using previous Python versions.



Important

Python provides various APIs for use with C extension modules. One of these APIs, `PyObject`, was deprecated in Python 2.7. By default, deprecation warnings are ignored so this will not normally cause any problems.

However, if the standard warning settings are overridden, there may be problems with modules that use `PyObject` and assume that the import succeeds. In particular, if warnings have been set to "error", it is possible to make the Python interpreter abort or even segfault when importing such modules due to reading through the NULL pointer triggered by the deprecation error.

To enable errors-for-warnings and use such a module, add an override so that a **`PendingDeprecationWarning`** is logged instead of raising an exception.

```
>>> import warnings
>>> warnings.simplefilter('error')
>>> warnings.simplefilter('default', PendingDeprecationWarning)
```

3.3.8.2. Python Debug Build

Red Hat Enterprise Linux 7 ships with a debug build of the python interpreter in the *python-debug* package.

The debug interpreter (found in `/usr/bin/python-debug`) runs at about half the speed as the optimized interpreter (found in `/usr/bin/python`) and requires extensions models to be rebuilt for it but is still of use when writing and debugging Python C extension modules. Within the debug build, optimization levels are turned down, making it easier to step through code within the debugger.

The debug build is configured with additional debug settings:

--with-pydebug

Adds various useful methods to `sys`, such as `sys.gettotalrefcount()` and `sys.getobjects()`.

--with-count-allocs

Enables the `COUNT_ALLOCS` setting, which adds a `sys.getcounts()` method, providing information on all types. The default upstream behavior is to always dump this information on `stdout` when the process exists. This is patched downstream so that the information is only dumped on exit if `PYTHONDUMPCOUNTS` is set in the environment.

--with-call-profile

Enables the `CALL_PROFILE` setting. This counts the number of function calls executed, and on how the interpreter handled those calls.

--with-tsc (only on x86_64 and ppc64)

Adds a `sys.settscdump()` method, adding very low-level profiling of the interpreter.

The debug build uses the same bytecode files as the regular optimized build, but extension modules (`.so` files) are not compatible. This is because the in-memory layout of Python objects differs due to the extra instrumentation. Given an optimized extension model `foo.so`, the debug build is patched to look for `foo_d.so`.

For more information on the debug build and its settings, see the notes upstream at <http://svn.python.org/projects/python/trunk/Misc/SpecialBuilds.txt>.

3.3.8.3. Python Documentation

For more information about Python, refer to `man python`. You can also install `python-docs`, which provides HTML manuals and references in the following location:

`file:///usr/share/doc/python-docs-version/html/index.html`

For details on library and language components, use `pydoc component_name`. For example, `pydoc math` will display the following information about the `math` Python module:

```

Help on module math:

NAME
  math

FILE
  /usr/lib64/python2.6/lib-dynload/mathmodule.so

DESCRIPTION
  This module is always available. It provides access to the
  mathematical functions defined by the C standard.

FUNCTIONS
  acos[...]
  acos(x)

  Return the arc cosine (measured in radians) of x.

  acosh[...]
  acosh(x)

  Return the hyperbolic arc cosine (measured in radians) of x.

  asin(...)
  asin(x)

  Return the arc sine (measured in radians) of x.

  asinh[...]
  asinh(x)

  Return the hyperbolic arc sine (measured in radians) of x.

```

The main site for the Python development project is hosted on python.org.

3.3.9. Java

Red Hat Enterprise Linux 7 is constantly updated to ship the latest version of JDK. The **java-version_number-openjdk** package adds support for the Java programming language. This package provides the **java** interpreter. The **java-version_number-openjdk-devel** package contains the **javac** compiler, as well as the libraries and header files required for developing Java extensions.

Red Hat Enterprise Linux also ships with numerous **java**-related packages. By convention, the names of these packages have a **java** prefix or suffix.

3.3.9.1. Java Features

Java has a number of new features with Red Hat Enterprise Linux 7. These include the following:

Support for dynamically-typed languages (InvokeDynamic)

Enhancements are made to Hotspot, Open JDK's Java Virtual Machine (JVM). These are designed to support dynamically typed languages with minimal performance cost as compared to statically typed languages and Java itself. Specifically, the `invokedynamic` instruction was added to the Java bytecode specification and implemented in the JVM.

Small language enhancements (Project Coin)

A number of Java language-level improvements that provide programmer conveniences, more elegant code, and reduces some common programming errors.

Strings in switch

In prior Java versions, `switch` statements allowed the use of `byte`, `short`, `char`, and `int` primitives and their corresponding object types, as well as `enums`. As of Java 7, string values may also be used in `switch` statements.

Binary integral literals and underscores in numeric literals

Programmers may now express integral literals in binary form, or separate groups of digits in numerical literal values by underscores, in order to improve code readability.

Multi-catch

Java's `catch` syntax has been improved so that more than one exception type can be caught in a single `catch` clause, reducing redundant code.

More precise rethrow

The Java 7 compiler has been improved so that a method that catches and then rethrows an exception can be more precise in the `throws` clause of the method declaration in some circumstances.

Improved type inference for generic instance creation (diamond)

This syntactical improvement allows programmers to use the diamond operator (that is, `<>`) instead of the full generic type (for example, `<ClassName>`) when instantiating variables of generic types. The type is inferred from that variable's declaration instead.

Try-with-resources statement

This is a new form of `try` statement for use with closeable resources, such as streams and files. Using this feature, programmers no longer need to explicitly close these resources.

Simplified varargs method invocation

Previously, a compiler warning would be issued when calling vararg methods with non-reifiable arguments. This has been removed and replaced with a warning at the declaration of a vararg method that can accept non-reifiable arguments. An annotation can be used to suppress this warning, in which case the developer takes responsibility that the arguments are correct. This primarily applies to vararg methods that accept generic arguments.

Concurrency and collections updates

A number of new classes and interfaces, including a fork/join framework for divide and conquer type algorithms, has been added to the `java.util.concurrent` package. These can be useful for improving performance and correctness of multi-threaded programs.

New I/O APIs for the Java platform

This includes a new file system API to improve cross-platform compatibility while making graceful failure handling easier for developers. It provides improved socket/channel API in the `java.nio.channels` package to remove unintuitive dependences on the `java.net` package. It also provides a new asynchronous I/O API.

Nimbus look and feel for swing

Informally introduced in Java 6 under the `com.sun.java.swing` package namespace, Nimbus has a vector-graphics based look and feel for swing. With Java 7, it has become an official API and moved to the `javax.swing` package.

3.3.9.2. Java Documentation

For more information about Java, refer to `man java`. Some associated utilities also have their own respective `man` pages.

You can also install other Java documentation packages for more details about specific Java utilities. By convention, such documentation packages have the `javadoc` suffix (for example, `dbus-java-javadoc`).

The main site for the development of Java is hosted on <http://openjdk.java.net/>. The main site for the library runtime of Java is hosted on <http://icedtea.classpath.org>.

3.3.10. Ruby

The `ruby` package provides the Ruby interpreter and adds support for the Ruby programming language. The `ruby-devel` package contains the libraries and header files required for developing Ruby extensions.

Red Hat Enterprise Linux also ships with numerous `ruby`-related packages. By convention, the names of these packages have a `ruby` or `rubygem` prefix or suffix. Such packages are either library extensions or Ruby bindings to an existing library.

Examples of `ruby`-related packages include:

- ▶ `ruby-irb`
- ▶ `ruby-libguestfs`
- ▶ `ruby-libs`
- ▶ `ruby-qpid`

- ▶ `ruby-rdoc`
- ▶ `ruby-ri`
- ▶ `ruby-tcltk`
- ▶ `rubygems`
- ▶ `rubygem-bigdecimal`
- ▶ `rubygem-devel`
- ▶ `rubygem-io-console`
- ▶ `rubygem-json`
- ▶ `rubygem-minitest`
- ▶ `rubygem-rake`



Note

If the Bundler is used for managing application dependencies, please always use the Bundler provided by the **rubygem-bundler** package. The upstream package is not compatible with the RubyGems layout used by Red Hat Enterprise Linux 7.

3.3.10.1. Ruby Updates

For information about updates to the Ruby language in Red Hat Enterprise Linux 7, refer to the following resources:

- ▶ `file:///usr/share/doc/ruby-version/NEWS`
- ▶ `file:///usr/share/doc/ruby-version/NEWS-version`

Ruby has undergone significant changes in its filesystem layout, which now better conforms with FHS. Binary libraries and extensions of Gems are placed under `/usr/lib` (or `/usr/lib64` for 64-bit systems) and pure Ruby libraries and Gems are placed under `/usr/share`. Gems are located in three places according to the selected method of their installation:

- ▶ `/usr`
- ▶ `/usr/local`
- ▶ `~/.gem`

3.3.10.2. Ruby Documentation

For more information about Ruby, refer to `man ruby`. You can also use the `ri` command, which is the Ruby API reference front end. For gem documentation, use the `gem server` command that makes HTML manuals and references about gems installed on your system available in a browser.



Note

It may be necessary to install the `-doc` sub-package to make the documentation available using the `ri` and `gem server` commands.

The main site for the development of Ruby is hosted on <http://www.ruby-lang.org>. The <http://www.ruby-doc.org> site also contains Ruby documentation. Online documentation for gems can be found at <http://rdoc.info>.

Documentation for the `ri` command can be found in `/usr/share/ri/system`.

3.3.11. Perl

The `perl` package adds support for the Perl programming language. This package provides some of the Perl core modules, the Perl Language Interpreter, and the `perldoc` tool. Red Hat Enterprise Linux 7 ships with `perl-5.16`. To install all of the core modules, use the `yum install perl-core` command.

Red Hat also provides various perl modules in package form; these packages are named with the `perl-*` prefix. These modules provide stand-alone applications, language extensions, Perl libraries, and external library bindings.

An RPM package can contain more Perl modules. Each module intended for public use is provided by the package in the form `perl(The::Module)`. This expression can be passed to `yum` to install the appropriate packages.

Example 3.1. Install perl module

```
# yum install 'perl(LWP::UserAgent)'
```

This will install the RPM package `perl-libwww-perl`, which contains the `LWP::UserAgent` module, allowing a programmer to use the command `use LWP::UserAgent;`

3.3.11.1. Perl Updates

Red Hat Enterprise Linux 7 ships with perl 5.16 which has a number of changes since the 5.10 version shipped in Red Hat Enterprise Linux 6. These include:

Perl 5.12 Updates

Perl 5.12 has the following updates:

- ▶ Perl conforms closer to the Unicode standard.
- ▶ Experimental APIs allow Perl to be extended with "pluggable" keywords and syntax.
- ▶ Perl will be able to keep accurate time well past the "Y2038" barrier.
- ▶ Package version numbers can be directly specified in "package" statements.
- ▶ Perl warns the user about the use of deprecated features by default.

The Perl 5.12 delta can be accessed at <http://perldoc.perl.org/perl5120delta.html>.

Perl 5.14 Updates

Perl 5.14 has the following updates:

- ▶ Unicode 6.0 support.
- ▶ Improved support for IPv6.
- ▶ Easier auto-configuration of the CPAN client.
- ▶ A new `/r` flag that makes `s///` substitutions non-destructive.
- ▶ New regular expression flags to control whether matched strings should be treated as ASCII or Unicode.
- ▶ New **package** *Foo* { } syntax.
- ▶ Less memory and CPU usage than previous releases.
- ▶ A number of bug fixes.

The Perl 5.14 delta can be accessed at <http://perldoc.perl.org/perl5140delta.html>.

Perl 5.16 Updates

Perl 5.16 has the following updates:

- ▶ Support for Unicode 6.1.
- ▶ `$$` variable is writable.
- ▶ Improved debugger.
- ▶ Accessing Unicode database files directly is now deprecated; use `Unicode::UCD` instead.
- ▶ `Version::Requirements` is deprecated in favor of `CPAN::Meta::Requirements`.
- ▶ A number of perl4 libraries are removed:
 - *abbrev.pl*
 - *assert.pl*
 - *bigfloat.pl*
 - *bigint.pl*
 - *bigrat.pl*
 - *cacheout.pl*
 - *complete.pl*
 - *ctime.pl*
 - *dotsh.pl*
 - *exceptions.pl*
 - *fastcwd.pl*

- *flush.pl*
- *getcwd.pl*
- *getopt.pl*
- *getopts.pl*
- *hostname.pl*
- *importenv.pl*
- *lib/find.pl*
- *lib/finddepth.pl*
- *look.pl*
- *newgetopt.pl*
- *open2.pl*
- *open3.pl*
- *pwd.pl*
- *hellwords.pl*
- *stat.pl*
- *tainted.pl*
- *termcap.pl*
- *timelocal.pl*

The Perl 5.16 delta can be accessed at <http://perldoc.perl.org/perl5160delta.html>.

3.3.11.2. Installation

Perl's capabilities can be extended by installing additional modules. These modules come in the following forms:

Official Red Hat RPM

The official module packages can be installed with **yum** or **rpm** from the Red Hat Enterprise Linux repositories. They are installed to **/usr/share/perl5** and either **/usr/lib/perl5** for 32bit architectures or **/usr/lib64/perl5** for 64bit architectures, as well as **vendor_perl** subdirectories.

Modules from CPAN

Use the **cpan** tool provided by the *perl-CPAN* package to install modules directly from the CPAN website. They are installed to **/usr/local/share/perl5** and either **/usr/local/lib/perl5** for 32bit architectures or **/usr/local/lib64/perl5** for 64bit architectures if these directories exist and are writable by the current user.

If the directories do not exist the **cpan** tool will offer different solutions.

```
Warning: You do not have write permission for Perl library directories.
```

```
To install modules, you need to configure a local Perl library directory or
escalate your privileges. CPAN can help you by bootstrapping the local::lib
module or by configuring itself to use 'sudo' (if available). You may also
resolve this problem manually if you need to customize your setup.
```

```
What approach do you want? (Choose 'local::lib', 'sudo' or 'manual')
[local::lib]
```

For example, if 'manual' is selected, it will assume the user will ensure the directories exist and are writable before installing modules from CPAN.

Third party and custom module packages

These packaged modules are installed to `/usr/share/perl5/vendor_perl` and either `/usr/lib/perl5/vendor_perl` for 32bit architectures or `/usr/lib64/perl5/vendor_perl` for 64bit architectures. If their file names conflict with Red Hat Enterprise Linux packages, either change the file names or properly replace the Red Hat Enterprise Linux packages with the delivering packages.



Warning

If an official version of a module is already installed, installing its non-official version can create conflicts in the `/usr/share/man` directory.

If an additional Perl module search path is necessary, the `/usr/local/share/perl5/sitecustomize.pl` script can be used for system-wide modification (see the `perlrun(1)` man page), or `perl-homedir` package for user specific modifications (see the `perl-homedir` package description).

3.3.11.3. Perl Documentation

The `perldoc` tool provides documentation on language and core modules. To learn more about a module, use `perldoc module_name`. For example, `perldoc CGI` will display the following information about the CGI core module:

```
NAME
CGI - Handle Common Gateway Interface requests and responses

SYNOPSIS
use CGI;

my $q = CGI->new;

[...]

DESCRIPTION
CGI.pm is a stable, complete and mature solution for processing and preparing HTTP
requests and responses. Major features including processing form submissions, file
uploads, reading and writing cookies, query string generation and manipulation, and
processing and preparing HTTP headers. Some HTML generation utilities are included
as well.

[...]
```

PROGRAMMING STYLE

There are two styles of programming with CGI.pm, an object-oriented style and a function-oriented style. In the object-oriented style you create one or more CGI objects and then use object methods to create the various elements of the page. Each CGI object starts out with the list of named parameters that were passed to your CGI script by the server.

[...]

For details on Perl functions, use **perldoc -f *function_name***. For example, `perldoc -f split` will display the following information about the `split` function:

```
split /PATTERN/,EXPR,LIMIT
```

```
split /PATTERN/,EXPR
```

```
split /PATTERN/
```

`split` Splits the string `EXPR` into a list of strings and returns that list. By default, empty leading fields are preserved, and empty trailing ones are deleted. (If all fields are empty, they are considered to be trailing.)

In scalar context, returns the number of fields found. In scalar and void context it splits into the `@_` array. Use of `split` in scalar and void context is deprecated, however, because it clobbers your subroutine arguments.

If `EXPR` is omitted, splits the `$_` string. If `PATTERN` is also omitted, splits on whitespace (after skipping any leading whitespace). Anything matching `PATTERN` is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.)

[...]

Current `perldoc` documentation can be found on perldoc.perl.org.

Core and external modules are documented on the [Comprehensive Perl Archive Network](http://ComprehensivePerlArchiveNetwork).

3.3.12. libStorageMgmt Plug-ins

Red Hat Enterprise Linux 7 ships with a new library called **libStorageMgmt**. It is a storage array independent Application Programming Interface (API) that provides a stable and consistent API allowing developers to programmatically manage different storage arrays and leverage the hardware accelerated features provided.

For more information on the **libStorageMgmt** library refer to Red Hat's *Storage Administration Guide*. This section details how to write plug-ins for the library.

Plug-ins work somewhat differently with the **libStorageMgmt** library. The plug-ins execute in their own address space as stand-alone executables with inter-process communication (IPC) between the client and plug-in. When a client application or the **libStorageMgmt** command line (**lsmcli**) utilizes the library, the following occurs:

1. The library uses the uniform resource identifier (URI) and parses out which plug-in was specified. For example, **LSMCLI_URI=sim://** refers to the simulator plug-in.
2. The library uses the plug-in name **sim** and looks for the unix domain socket in the socket directory. The default directory is **/var/run/lsm/ipc** but this can be changed at run-time by specifying the **LSM_UDS_PATH** environment variable.

3. The client library opens the unix domain socket, causing the **lsmd** daemon to accept the connection from the client. The daemon then forks and executes the plug-in, passing the socket descriptor on the command line to the plug-in. The client process now has a direct connection to the plug-in.
4. The **lsmd** is no longer in the path and goes back to sleep waiting for another process to open a socket.

There are a number of benefits to this different design. These include:

- ▶ If a daemon dies or is killed, any existing client plug-in sessions remain active.
- ▶ If a plug-in crashes, the client process will remain operational.
- ▶ The daemon needs to know nothing of the IPC protocol, keeping it simple.
- ▶ The plug-in can be closed source if required by the vendor.

3.3.12.1. Writing a plug in for **libStorageMgmt** library

The **libStorageMgmt** library has a plug-in API for both C and Python. Any language that supports sockets and text can also be used to write a plug-in, but the library provides the abstraction that hides this complexity.

The following are some general guidelines for plug-in design regardless of the programming language used:

Threading or multi-process

The library does not provide locking, nor does it keep any global state. As such, it is valid for a client to have a separate plug-in instance in use for each thread or process. Plug-ins can anticipate that multiple instances of themselves can and possibly will be running at concurrently to different arrays. As the library provides a mechanism for long-running operations, multiple plug-in instances for the same array are not needed.

Plug-ins execute with non-root privileges

To reduce the potential for local exploits, plug-ins have reduced privileges. This needs to be taken into account when writing and designing plug-ins.

Plug-in lifetime

The client API provides for a handle that is opened and closed for each plug-in instance. During this time the plug-in is free to cache whatever data is necessary to provide correct operation. When using the **lsmdi** tool, the lifetime is only for one command.

Logging

Plug-ins log errors to **syslog**. Helper functions exist to facilitate this in the library.

Errors

The library uses well defined error codes in order to remain language agnostic. Additional error data can be retrieved when they occur to provide textual error messages and optionally debug data from the plug-in or the array itself. It is the library callers' responsibility to retrieve this additional information after an error occurs and before issuing another command. If additional error data exists and other functions are called, then the additional error information will be lost. C does not support exceptions. For languages that do support exceptions, a custom exception class containing the error code and additional information is provided.

Location and naming

Plug-ins are located in the `/usr/bin` directory. The name format must be `_lsmplugin`. This is because when the daemon starts it iterates in the directory enumerating them.

Job Control

The methods to get and set the time-out are used to specify how long the plug-in waits for a response from the array. If an operation can not safely complete within the time-out, the call returns a job id so that the client can check on the status of the operation. Job IDs are free form strings and are plug-in defined. The plug-in implementation needs to determine everything about the asynchronous operation from this string between invocations of the plug-in.

To write a plug-in, the following base functions or methods are required for all plug-ins, regardless of the language used:

- ▶ get/set timeout
- ▶ startup/shutdown
- ▶ job status
- ▶ job free
- ▶ capabilities
- ▶ plug-in information
- ▶ pools
- ▶ systems

A unique name must also be chosen so that the main executable has the form `name_lsmplugin`.

The following sections detail how to write a plug-in for python and for C.

3.3.12.1.1. Writing a plug-in with Python

First, implement the interface that supports the level of functionality to be provided (refer to `iplugin.py`). Most plug-ins will either inherit from `IStorageAreaNetwork` or `INfs`, or both if the plug-in supports block and network file systems.

Next, call the plug-in runner, passing the name of the class and the command line arguments to it for processing and executing the run method.

```
#!/usr/bin/env python
import sys

from lsm.pluginrunner import PluginRunner
from lsm.simulator import StorageSimulator

if __name__ == '__main__':
    PluginRunner(StorageSimulator, sys.argv).run()
```

**Note**

During development it is possible to call the plug-in directly on the command line for easier debugging.

3.3.12.1.2. Writing a plug-in with C

First, include the required header file `#include <libstoragemgmt/libstoragemgmt_plug_interface.h>`.

Then, implement the callback functions that will be supported, along with the required ones.

Finally, pass the command line count and arguments to the library with load and unload functions.

```
#include <libstoragemgmt/libstoragemgmt_plug_interface.h>
#include <stdlib.h>
#include <stdint.h>

static char name[] = "Simple limited plug-in example";
static char version [] = "0.01";

struct plugin_data {
    uint32_t tmo;
    /* All your other variables as needed */
};

/* Create the functions you plan on implementing that
match the callback signatures */
static int tmoSet(lsm_plugin_ptr c, uint32_t timeout, lsm_flag flags )
{
    int rc = LSM_ERR_OK;
    struct plugin_data *pd = (struct plugin_data*)lsm_private_data_get(c);
    /* Do something with state to set timeout */
    pd->tmo = timeout;
    return rc;
}

static int tmoGet(lsm_plugin_ptr c, uint32_t *timeout, lsm_flag flags )
{
    int rc = LSM_ERR_OK;
    struct plugin_data *pd = (struct plugin_data*)lsm_private_data_get(c);
    /* Do something with state to get timeout */
    *timeout = pd->tmo;
    return rc;
}

/* Setup the function addresses in the appropriate
required callback structure */
static struct lsm_mgmt_ops_v1 mgmOps = {
    tmoSet,
    tmoGet,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};
```

```

int load( lsm_plugin_ptr c, const char *uri, const char *password,
          uint32_t timeout, lsm_flag flags )
{
    /* Do plug-in specific init. and setup callback structures */
    struct plugin_data *data = (struct plugin_data *)
        malloc(sizeof(struct plugin_data));

    if (!data) {
        return LSM_ERR_NO_MEMORY;
    }

    /* Call back into the framework */
    int rc = lsm_register_plugin_v1( c, data, &mgmOps, NULL, NULL, NULL);
    return rc;
}

int unload( lsm_plugin_ptr c, lsm_flag flags)
{
    /* Get a handle to your private data and do clean-up */
    struct plugin_data *pd = (struct plugin_data*)lsm_private_data_get(c);
    free(pd);
    return LSM_ERR_OK;
}

int main(int argc, char *argv[] )
{
    return lsm_plugin_init_v1(argc, argv, load, unload, name, version);
}

```

3.3.12.2. Writing Plug-in References

- ▶ The python simulated plug-in sim <https://sourceforge.net/p/libstoragemgmt/code/ci/master/tree/lsm/lsm/simulator.py>
- ▶ The libStorageMgmt Writing Plug-ins wiki <https://sourceforge.net/p/libstoragemgmt/wiki/WritingPlugins/>

[1] MPI support is not available on IBM System Z machines (where Open MPI is not available).

Chapter 4. Compiling and Building

Red Hat Enterprise Linux includes many packages used for software development, including tools for compiling and building source code. This chapter discusses several of these packages and tools used to compile source code.

4.1. GNU Compiler Collection (GCC)

The GNU Compiler Collection (GCC) is a set of tools for compiling a variety of programming languages (including C, C++, ObjectiveC, ObjectiveC++, Fortran, and Ada) into highly optimized machine code. These tools include various compilers (like **gcc** and **g++**), run-time libraries (like **libgcc**, **libstdc++**, **libgfortran**, and **libgomp**), and miscellaneous other utilities.

Red Hat Enterprise Linux 7 is on PPC64 architecture. This means that GCC generates code for POWER7 platforms with tuning to POWER7 by default. The following tables detail the default codes for various platforms:

Table 4.1. Default Codes for i?86

Option	Default Code
mtune	generic
march	x86-64

redhat-rpm-config, used for building packages, also contains the **-mfpmath=sse** option.

Table 4.2. Default Codes for x86_64

Option	Default Code
mtune	generic
march	x86-64

Table 4.3. Default Codes for s390 and s390x

Option	Default Code
mtune	zEC12
march	z196

Table 4.4. Default Codes for ppc, ppc64, and ppc64p7

Option	Default Code
mtune	power7
mcpu	power7

The above table is for both 32-bit and 64-bit compilations.

It is possible to generate code for other targets using the **-march=CPU** option. Tuning the code for a specific chip is performed by using the **-mtune=CPU** option. The **-march** option implies the **-mtune** option. The **-march=native** option instructs the compiler to generate code for the processor type of the compiling machine.



Note

The `-march` option does not exist on PPC64 architecture. Use the `-mcpu` option instead.

4.1.1. Changes in GCC

Red Hat Developer Toolset 2.0 is distributed with **GCC 4.8**, which provides a number of bug fixes and feature enhancements over the Red Hat Enterprise Linux system version and the version included in Red Hat Developer Toolset 1.1. Below is a comprehensive list of new features and compatibility changes in this release.

4.1.1.1. Changes Since Red Hat Developer Toolset 1.1

The following features have been added since the release of GCC included in Red Hat Developer Toolset 1.1.

4.1.1.1.1. Caveats

Aggressive Loop Optimizations

The loop optimizer of GCC has been improved to use language constraints in order to derive bounds for the number of iterations of a loop. The bounds are then used as a guide to loop unrolling, peeling, and loop exit test optimizations.

The optimizations assume that the loop code does not invoke undefined behavior by, for example, causing signed integer overflows or making out-of-bound array accesses. For example, consider the following code fragment:

```

unsigned int foo()
{
    unsigned int data_data[128];

    for (int fd = 0; fd < 128; ++fd)
        data_data[fd] = fd * (0x02000001); // error

    return data_data[0];
}

```

When the value of the `fd` variable is 64 or above, the `fd * 0x02000001` operation overflows, which is invalid in both C and C++ for signed integers. In the example above, GCC may generate incorrect code or enter an infinite loop.

To fix this error, use the appropriate casts when converting between signed and unsigned types to avoid overflows, for instance:

```

data_data[fd] = (uint32_t) fd * (0x02000001U); // ok

```

If necessary, this optimization can be turned off by using the new command line option `-fno-aggressive-loop-optimizations`.

4.1.1.1.2. General Improvements and Changes

New Local Register Allocator

GCC 4.8 features a new *Local Register Allocator* (LRA), which replaces the 26-year old reload pass and improves the quality of generated code. The new local register allocator is meant to be simpler, easier to debug, and does a better job of register allocation.

AddressSanitizer

A fast memory error detector called *AddressSanitizer* has been added and can be enabled by using the `-fsanitize=address` command line option. It augments memory access instructions in order to detect use-after-free and out-of-bound accesses to objects on the heap.

ThreadSanitizer

A fast data race detector called *ThreadSanitizer* has been added in GCC 4.8. The option to enable this feature is `-fsanitize=thread`.

Compiling Extremely Large Functions

Many scalability bottlenecks have been removed from GCC optimization passes. As a consequence, it is now possible to compile extremely large functions with smaller memory consumption in less time.

New -Og Optimization Level

A new general optimization level, `-Og`, has been introduced. This optimization level addresses the need for fast compilation and a superior debugging experience while providing a reasonable level of runtime performance. Overall, the development experience should be better than the default optimization level `-O0`.

Caret Diagnostic Messages

The diagnostic messages of GCC, which display a line of source code, now also show a caret that indicates the column where the problem was detected. For example:

```
fred.cc:4:15: fatal error: foo: No such file or directory
#include <foo>
           ^
compilation terminated.
```

New -fira-hoist-pressure Option

A new command line option, `-fira-hoist-pressure`, has been added. This option uses the register allocator to help decide when it is worthwhile to move expressions out of loops. It can reduce the size of the compiler code, but it slows down the compiler. This option is enabled by default at `-Os`.

New -fopt-info Option

A new command line option, `-fopt-info`, has been added. This option controls printing information about the effects of particular optimization passes, and takes the following form:

```
-fopt-info[-info][=file_name]
```

The *info* part of the option controls what is printed. Replace it with **optimized** to print information when optimization takes place, **missed** to print information when optimization does not take place, **note** to print more verbose information, or **optall** to print everything.

Replace *file_name* with the name of the file in which you want the information to be written. If you omit this part of the option, GCC writes the information to the standard error output stream.

For example, to display a list of optimizations that were enabled by the **-O2** option but had no effect when compiling a file named **foo.c**, type:

```
gcc -O2 -fopt-info-missed foo.c
```

New -floop-nest-optimize Option

A new command line option, **-floop-nest-optimize**, has been added. This option enables an experimental ISL-based loop nest optimizer, a generic loop nest optimizer that is based on the Pluto optimization algorithms and that calculates a loop structure optimized for data-locality and parallelism. For more information about this optimizer, see <http://pluto-compiler.sourceforge.net>.

Hot and Cold Attributes on Labels

The hot and cold function attributes can now also be applied to labels. Hot labels tell the compiler that the execution path following the label is more likely than any other execution path, and cold labels convey the opposite meaning. These attributes can be used in cases where **__builtin_expect** cannot be used, for instance with a computed **goto** or **asm goto**.

4.1.1.1.3. Debugging Enhancements

DWARF4

DWARF4 is now used as the default debugging data format when generating debugging information. To get the maximum benefit from this new debugging representation, use the latest version of **Valgrind**, **elfutils**, and **GDB** included in this release.

New -gsplit-dwarf Option

A new command line option, **-gsplit-dwarf**, has been added. This option tells the compiler driver to separate as much DWARF debugging information as possible into a separate output file with the **.dwo** file extension, and allows the build system to avoid linking files with debugging information.

In order to be useful, this option requires a debugger capable of reading **.dwo** files, such as the version of **GDB** included in Red Hat Developer Toolset 2.0.



Note

elfutils, **SystemTap**, and **Valgrind** do *not* support the **.dwo** files.

4.1.1.1.4. C++ Changes

Experimental C++ Features from an Upcoming Standard

g++ now supports a new command line option, **-std=c++1y**. This option can be used for experimentation with features proposed for the next revision of the standard that is expected around 2014. Currently, the only difference from **-std=c++11** is support for return type deduction in normal functions as proposed in [N3386](http://ericniebler.com/2014/01/24/range-comprehensions/).

New `thread_local` Keyword

`g++` now implements the C++11 `thread_local` keyword. In comparison with the GNU `__thread` keyword, `thread_local` allows dynamic initialization and destruction semantics.

The use of the `thread_local` keyword has currently one important limitation: when the `dlclose()` function is used to unload a dynamically loaded DSO that contains the definition of a `thread_local` object, the `thread_local` object is destroyed, its destructor is called and the DSO is unmapped from the address space of the process. If a thread in the process tries to access the `thread_local` object after this, the program may terminate unexpectedly. As a result, the programmer may have to take extra care to ensure that `thread_local` objects in a DSO are not referred after it has been unloaded.

See also the next item for dynamic initialization issues.

Dynamic Initialization of Thread-local Variables

The C++11 and OpenMP standards allow thread-local and thread-private variables to have dynamic (that is, runtime) initialization. To support this, any use of such a variable goes through a wrapper function that performs necessary initialization.

When the use and definition of the variable are in the same translation unit, this overhead can be optimized away, but when the use is in a different translation unit, there is significant overhead even if the variable does not actually need dynamic initialization. If the programmer can be sure that no use of the variable in a non-defining translation unit needs to trigger dynamic initialization (either because the variable is statically initialized, or a use of the variable in the defining translation unit will be executed before any uses in another translation unit), they can avoid this overhead by using the new `-fno-extern-tls-init` option.

By default, `g++` uses the `-fextern-tls-init` option.

C++11 Attribute Syntax

`g++` now implements the C++11 attribute syntax, for example:

```
[[noreturn]] void f();
```

C++11 Alignment Specifier

`g++` now implements the C++11 alignment specifier, for example:

```
alignas(double) int i;
```

4.1.1.1.5. Fortran Changes

4.1.1.1.5.1. Caveats

The version of module files (the `.mod` files) has been incremented. Fortran modules compiled by earlier GCC versions have to be recompiled when they are used by files compiled with GCC 4.8, as this version of GCC is not able to read `.mod` files created by earlier versions; attempting to do so fails with an error message.

**Note**

The ABI of the produced assembler data itself has not changed; object files and libraries are fully compatible with older versions except as noted in [Section 4.1.1.1.5.2, “ABI Compatibility”](#).

4.1.1.1.5.2. ABI Compatibility

Some internal names used in the assembler or object file have changed for symbols declared in the specification part of a module. If an affected module — or a file using it via use association — is recompiled, the module and all files which directly use such symbols have to be recompiled as well. This change only affects the following kind of module symbols:

- ▶ *Procedure pointers.* Note that C-interoperable function pointers (**type(c_funptr)**) are not affected, nor are procedure-pointer components.
- ▶ *Deferred-length character strings.*

4.1.1.1.5.3. Other Changes**BACKTRACE Intrinsic**

A new intrinsic subroutine, **BACKTRACE**, has been added. This subroutine shows a backtrace at an arbitrary place in user code, program execution continues normally afterwards.

Floating Point Numbers with “q” as Exponential

Reading floating point numbers that use **q** for the exponential (such as **4.0q0**) is now supported as a vendor extension for better compatibility with old data files. It is strongly recommended to use the equivalent but standard conforming **e** (such as **4.0e0**) for I/O.

For Fortran source code, consider replacing the **q** in floating-point literals by a kind parameter (such as **4.0e0_qp** with a suitable **qp**). Note that — in Fortran source code — replacing **q** with a simple **e** is not equivalent.

GFORTTRAN_TMPDIR Environment Variable

The **GFORTTRAN_TMPDIR** environment variable for specifying a non-default directory for files opened with **STATUS="SCRATCH"**, is not used anymore. Instead, **gfortran** checks the POSIX/GNU standard **TMPDIR** environment variable and if **TMPDIR** is not defined, **gfortran** falls back to other methods to determine the directory for temporary files as documented in the user manual.

Fortran 2003

Support for unlimited polymorphic variables (**CLASS(*)**) has been added. Non-constant character lengths are not yet supported.

TS 29113

Assumed types (**TYPE(*)**) are now supported.

Experimental support for assumed-rank arrays (**dimension(. .)**) has been added. Note that at the moment, the **gfortran** array descriptor is used, which is different from the array descriptor defined in *TS 29113*. For more information, see the header file of **gfortran** or use the **Chasm** language interoperability tools.

4.1.1.1.6. x86-specific Improvements

New Instructions

GCC 4.8 has added support for the Intel **FXSR**, **XSAVE**, and **XSAVEOPT** instructions. Corresponding intrinsics and built-in functions can now be enabled by using the **-mfxsr**, **-mxsave**, and **-mxsaveopt** command line options respectively.

In addition, support for the **RDSEED**, **ADCX**, **ADOX**, and **PREFETCHW** instructions has been added and can be enabled by using the **-mrdseed**, **-madx**, and **-mprfchw** command line options.

New Built-in Functions to Detect Run-time CPU Type and ISA

A new built-in function, **__builtin_cpu_is()**, has been added to detect if the run-time CPU is of a particular type. This function accepts one string literal argument with the CPU name, and returns a positive integer on a match and zero otherwise. For example, **__builtin_cpu_is("westmere")** returns a positive integer if the run-time CPU is an Intel Core i7 Westmere processor. For a complete list of valid CPU names, see the user manual.

A new built-in function, **__builtin_cpu_supports()**, has been added to detect if the run-time CPU supports a particular ISA feature. This function accepts one string literal argument with the ISA feature, and returns a positive integer on a match and zero otherwise. For example, **__builtin_cpu_supports("sse3")** returns a positive integer if the run-time CPU supports **SSSE3** instructions. For a complete list of valid ISA names, see the user manual.



Important

If these built-in functions are called before any static constructors are invoked, such as **IFUNC** initialization, then the CPU detection initialization must be explicitly run using this newly provided built-in function, **__builtin_cpu_init()**. The initialization needs to be done only once. For example, the following is sample invocation inside an **IFUNC** initializer:

```
static void (*some_ifunc_resolver(void))(void)
{
    __builtin_cpu_init();
    if (__builtin_cpu_is("amdfam10h") ...
    if (__builtin_cpu_supports("popcnt") ...
}
```

Function Multiversioning

Function multiversioning allows the programmer to specify multiple versions of the same function, each of which is specialized for a particular variant of a given target. At runtime, the appropriate version is automatically executed depending upon the target where the execution takes place. For example, consider the following code fragment:

```
__attribute__((target ("default"))) int foo () { return 0; }
__attribute__((target ("sse4.2"))) int foo () { return 1; }
__attribute__((target ("arch=atom"))) int foo () { return 2; }
```

When the function **foo()** is executed, the result returned depends upon the architecture where the program runs, not the architecture where the program was compiled. See the [GCC Wiki](#) for more details.

New RTM and HLE Ininsics

Support for the Intel RTM and HLE intrinsics, built-in functions, and code generation has been added and can be enabled by using the `-mrtm` and `-mhle` command line options. This is done via intrinsics for *Restricted Transactional Memory* (RTM) and extensions to the memory model for *Hardware Lock Elision* (HLE).

For HLE, two new flags can be used to mark a lock as using hardware elision:

`__ATOMIC_HLE_ACQUIRE`

Starts lock elision on a lock variable. The memory model in use must be `__ATOMIC_ACQUIRE` or stronger.

`__ATOMIC_HLE_RELEASE`

Ends lock elision on a lock variable. The memory model must be `__ATOMIC_RELEASE` or stronger.

For example, consider the following code fragment:

```
while (__atomic_exchange_n (& lockvar, 1, __ATOMIC_ACQUIRE
                           | __ATOMIC_HLE_ACQUIRE))
    _mm_pause ();
// work with the acquired lock
__atomic_clear (& lockvar, __ATOMIC_RELEASE | __ATOMIC_HLE_RELEASE);
```

The new intrinsics that support Restricted Transactional Memory are:

`unsigned _xbegin (void)`

Attempts to start a transaction. If it succeeds, this function returns `_XBEGIN_STARTED`, otherwise it returns a status value indicating why the transaction could not be started.

`void _xend (void)`

Commits the current transaction. When no transaction is active, this function causes a fault. All memory side effects of the transactions become visible to other threads in an atomic manner.

`int _xtest (void)`

Returns a non-zero value if a transaction is currently active, or zero if it is not.

`void _xabort (unsigned char status)`

Aborts the current transaction. When no transaction is active, this is a no-op. The parameter `status` is included in the return value of any `_xbegin()` call that is aborted by this function.

The following example illustrates the use of these intrinsics:

```
if ((status = _xbegin ()) == _XBEGIN_STARTED)
{
    // some code
    _xend ();
}
```

```
else
{
    // examine the status to see why the transaction failed and possibly retry
}
```

Transactions Using Transactional Synchronization Extensions

Transactions in the transactional memory feature (the `-fgnu-tm` option) of GCC can now be run using *Transactional Synchronization Extensions* (TSX) if available on x86 hardware.

Support for AMD Family 15h Processors

The x86 backend of GCC now supports CPUs based on AMD Family 15h cores with the 64-bit x86 instruction set support. This can be enabled by using the `-march=bdver3` option.

Support for AMD Family 16h Processors

The x86 backend of GCC now supports CPUs based on AMD Family 16h cores with the 64-bit x86 instruction set support. This can be enabled by using the `-march=btver2` option.

4.1.1.2. Changes Since Red Hat Enterprise Linux 6.4 and 5.9

The following features have been added since the release of GCC included in Red Hat Enterprise Linux 6.4 and 5.9:

4.1.1.2.1. Status and Features

4.1.1.2.1.1. C++11

GCC 4.7 and later provides experimental support for building applications compliant with C++11 using the `-std=c++11` or `-std=gnu++11` command line options. However, there is no guarantee for compatibility between C++11 code compiled by different versions of the compiler. Refer to [Section 4.1.1.2.3.1, "C++ ABI"](#) for details.

The C++ runtime library, `libstdc++`, supports a majority of the C++11 features. However, there is no or only partial support for some features such as certain properties on type traits or regular expressions. For details, refer to the [libstdc++ documentation](#), which also lists implementation-defined behavior.

Support for C++11 `exception_ptr` and `future` requires changes to the exception handling runtime in the system `libstdc++` package. These changes will be distributed through the normal Z-stream channel. Application of all Red Hat Enterprise Linux errata may be required to see correct runtime functionality when using these features.

4.1.1.2.1.2. C11

GCC 4.7 and later provides experimental support for some of the features from the C11 revision of the ISO C standard, and in addition to the previous (now deprecated) `-std=c1x` and `-std=gnu1x` command line options, gcc now accepts `-std=c11` and `-std=gnu11`. Note that since this support is experimental, it may change incompatibly in future releases.

Examples for features that are supported are Unicode strings (including the predefined macros `__STDC_UTF_16__` and `__STDC_UTF_32__`), nonreturning functions (`_Noreturn` and `<stdnoreturn.h>`), and alignment support (`_Alignas`, `_Alignof`, `max_align_t`, and `<stdalign.h>`).

4.1.1.2.1.3. Parallelism and Concurrency

GCC 4.7 and later provides improved support for programming parallel applications:

1. The GCC compilers support the OpenMP API specification for parallel programming, version 3.1. Refer to the [OpenMP](#) website for more information about this specification.
2. The C++11 and C11 standards provide programming abstractions for multi-threaded programs. The respective standard libraries include programming abstractions for threads and thread-related features such as locks, condition variables, or futures. These new versions of the standard also define a memory model that precisely specifies the runtime behavior of a multi-threaded program, such as the guarantees provided by compilers and the constraints programmers have to pay attention to when writing multi-threaded programs.

Note that support for the memory model is still experimental (see below for details). For more information about the status of support for C++11 and C11, refer to [Section 4.1.1.2.1.1, “C++11”](#) and [Section 4.1.1.2.1.2, “C11”](#) respectively.

The rest of this section describes two new GCC features in more detail. Both these features make it easier for programmers to handle concurrency (such as when multiple threads do not run truly in parallel but instead have to synchronize concurrent access to shared state), and both provide atomicity for access to memory but differ in their scope, applicability, and complexity of runtime support.

C++11 Types and GCC Built-ins for Atomic Memory Access

C++11 has support for *atomic types*. Access to memory locations of this type is atomic, and appears as one indivisible access even when other threads access the same memory location concurrently. The atomicity is limited to a single read or write access or one of the other atomic operations supported by such types (for example, two subsequent operations executed on a variable of atomic type are each atomic separately, but do not form one joint atomic operation).

An atomic type is declared as `atomic<T>`, where *T* is the non-atomic base type and must be trivially copyable (for example, `atomic<int>` is an atomic integer). GCC does not yet support any base type *T*, but only those that can be accessed atomically with the atomic instructions offered by the target architecture. This is not a significant limitation in practice, given that atomics are primarily designed to expose hardware primitives in an architecture-independent fashion; pointers and integrals that are not larger than a machine word on the target are supported as base types. Using base types that are not yet supported results in link-time errors.

The code generated for operations on atomic types, including the memory orders, implements the semantics specified in the C++11 standard. However, support for the C++11 memory model is still experimental, and for example GCC might not always preserve data-race freedom when optimizing code.

GCC also supports new built-ins for atomic memory accesses, which follow the design of the memory model and new atomic operations. The former set of synchronization built-ins (that is, those prefixed with `__sync`) are still supported.

Transactional Memory

Transactional Memory (TM) allows programs to declare that a piece of code is supposed to execute as a transaction, that is, virtually atomically and in isolation from other transactions. GCC's transactional memory runtime library, `libitm`, then ensures this atomicity guarantee when executing the compiled program. Compared to atomic memory accesses, it is a higher-level programming abstraction, because it is not limited to single memory locations, does not require special data types for the data it modifies, and because transactions can contain arbitrary code and be nested within other transactions (with some restrictions explained subsequently).

GCC implements transactions as specified in the [Draft Specification for Transactional Language Constructs for C++, version 1.1](#). This draft does not yet specify the language constructs for C, but GCC already supports a C-compatible subset of the constructs when compiling C source code.

The main language constructs are transaction statements and expressions, and are declared by the `__transaction_atomic` or `__transaction_relaxed` keywords followed by a compound statement or expression, respectively. The following example illustrates how to increment a global variable `y` if another variable `x` has a value less than 10:

```
__transaction_atomic { if (x < 10) y++; }
```

This happens atomically even in a multi-threaded execution of the program. In particular, even though the transaction can load `x` and `y` and store to `y`, all these memory accesses are virtually executed as one indivisible step.

Note that in line with the C++11 memory model, programs that use transactions must be free of data races. Transactions are guaranteed to be virtually executed serially in a global total order that is determined by the transactional memory implementation and that is consistent with and contributes to the happens-before order enforced by the rest of the program (that is, transaction semantics are specified based on the C++11 memory model, see the draft specification linked above). Nonetheless, if a program is not data-race-free, then it has undefined behavior. For example, a thread can first initialize some data and then make it publicly accessible by code like this:

```
init(data);
__transaction_atomic { data_public = true;} // data_public is initially false
```

Another thread can then safely use the data, for instance:

```
__transaction_atomic { if (data_public) use(data); }
```

However, the following code has a data race and thus results in undefined behavior:

```
__transaction_atomic { temp = copy(data); if (data_public) use(temp); }
```

Here, `copy(data)` races with `init(data)` in the initializing thread, because this can be executed even if `data_public` is not true. Another example for data races is one thread accessing a variable `x` transactionally and another thread accessing it nontransactionally at potentially the same time. Note that the data can be safely reclaimed using code like this (assuming only one thread ever does this):

```
__transaction_atomic { data_public = false; }
destruct(data);
```

Here, `destruct()` does not race with potential concurrent uses of the data because after the transaction finishes, it is guaranteed that `data_public` is false and thus data is private. See the specification and the C++11 memory model for more background information about this.

Note that even if transactions are required to virtually execute in a total order, this does not mean that they execute mutually exclusive in time. Transactional memory implementations attempt to run transactions as much in parallel as possible to provide scalable performance.

There are two variants of transactions: *atomic transactions* (`__transaction_atomic`) and *relaxed transactions* (`__transaction_relaxed`). The former guarantee atomicity with regard to all other code, but allow only code that is known to not include nontransactional kinds of synchronization, such as atomic or volatile memory access. In contrast, relaxed transactions allow all code (for example calls to I/O

functions), but only provide atomicity with regard to other transactions. Therefore, atomic transactions can be nested within other atomic and relaxed transactions, but relaxed transactions can only be nested within other relaxed transactions. Furthermore, relaxed transactions are likely to be executed with less performance, but this depends on the implementation and available hardware.

GCC verifies these restrictions statically at compile time (for example, the requirements on code allowed to be called from within atomic transactions). This has implications for when transactions call functions that are defined within other compilation unit (source file) or within libraries. To enable such cross-compilation-unit calls for transactional code, the respective functions must be marked to contain code that is safe to use from within atomic transactions. Programmers can do so by adding the `transaction_safe` function attribute to the declarations of these functions and by including this declaration when defining the function. In turn, GCC then verifies that the code in these functions is safe for atomic transactions and generates code accordingly. If the programmer does not follow these constraints and/or steps, compile-time or link-time errors occur. Note that within a compilation unit, GCC detects automatically whether a function is safe for use within transactions, and the attributes therefore typically do not need to be added. See the draft specification linked above for further details.

GCC's transactional memory support is designed in such a way that it does not decrease the performance of programs that do not use transactions, nor the performance of nontransactional code, except due to the normal kinds of interference by concurrent threads that use the same resources such as the CPU.

Transactional memory support in GCC and `libitm` is still experimental, and both the ABI and API could change in the future if this is required due to the evolution of the specification of the language constructs, or due to implementation requirements. Note that when executing applications built with the `-fgnu-tm` command line option, it is currently a prerequisite to also have the appropriate version of the `libitm.so.1` shared library installed.

4.1.1.2.1.4. Architecture-specific Options

Red Hat Developer Toolset 2.0 is only available for Red Hat Enterprise Linux 5 and 6, both for the 32-bit and 64-bit Intel and AMD architectures. Consequently, the options described below are only relevant to these architectures.

Optimization for several processors is now available through the command line options described in [Table 4.5, “Processor Optimization Options”](#).

Table 4.5. Processor Optimization Options

Option	Description
<code>-march=core2</code> and <code>-mtune=core2</code>	Optimization for Intel Core 2 processors.
<code>-march=corei7</code> and <code>-mtune=corei7</code>	Optimization for Intel Core i3, i5, and i7 processors.
<code>-march=corei7-avx</code> and <code>-mtune=corei7-avx</code>	Optimization for Intel Core i3, i5, and i7 processors with AVX.
<code>-march=core-avx-i</code>	Optimization for the Intel processor code-named Ivy Bridge with RDRND, FSGSBASE, and F16C.
<code>-march=core-avx2</code>	Optimization for a next-generation processor from Intel with AVX2, FMA, BMI, BMI2, and LZCNT.
<code>-march=bdver2</code> and <code>-mtune=bdver2</code>	Optimization for AMD Opteron processors code-named Piledriver.
<code>-march=btver1</code> and <code>-mtune=btver1</code>	Optimization for AMD family 14 processors code-named Bobcat.
<code>-march=bdver1</code> and <code>-mtune=bdver1</code>	Optimization for AMD family 15h processors code-named Bulldozer.

Support for various processor-specific intrinsics and instructions is now available through the command line options described in [Table 4.6, “Support for Processor-specific Intrinsics and Instructions”](#).

Table 4.6. Support for Processor-specific Intrinsics and Instructions

Option	Description
-mavx2	Support for Intel AVX2 intrinsics, built-in functions, and code generation.
-mbmi2	Support for Intel BMI2 intrinsics, built-in functions, and code generation.
-mlzcnt	Implementation and automatic generation of <code>__builtin_clz*</code> using the <code>lzcnt</code> instruction.
-mfma	Support for Intel FMA3 intrinsics and code generation.
-mfsgsbase	Enables the generation of new segment register read/write instructions through dedicated built-ins.
-mrdrnd	Support for the Intel <code>rdrnd</code> instruction.
-mf16c	Support for two additional AVX vector conversion instructions.
-mtbm	Support for TBM (Trailing Bit Manipulation) built-in functions and code generation.
-mbmi	Support for AMD's BMI (Bit Manipulation) built-in functions and code generation.
-mcrc32	Support for <code>crc32</code> intrinsics.
-mmovbe	Enables the use of the <code>movbe</code> instruction to implement <code>__builtin_bswap32</code> and <code>__builtin_bswap64</code> .
-mxop, -mfma4, and -mlwp	Support for the XOP, FMA4, and LWP instruction sets for the AMD Orochi processors.
-mabm	Enables the use of the <code>popcnt</code> and <code>lzcnt</code> instructions on AMD processors.
-mpopcnt	Enables the use of the <code>popcnt</code> instruction on both AMD and Intel processors.

When using the x87 floating-point unit, GCC now generates code that conforms to ISO C99 in terms of handling of floating-point excess precision. This can be enabled by `-fexcess-precision=standard` and disabled by `-fexcess-precision=fast`. This feature is enabled by default when using standards conformance options such as `-std=c99`.

Vectors of type `vector long long` or `vector long` are passed and returned using the same method as other vectors with the `VSX` instruction set. Previously GCC did not adhere to the ABI for 128-bit vectors with 64-bit integer base types (see GCC PR 48857).

The `-mrecip` command line option has been added, which indicates whether the reciprocal and reciprocal square root instructions should be used.

The `-mveclibabi=mass` command line option has been added. This can be used to enable the compiler to auto-vectorize mathematical functions using the Mathematical Acceleration Subsystem library.

The `-msingle-pic-base` command line option has been added, which instructs the compiler to avoid loading the `PIC` base register in function prologues. The `PIC` base register must be initialized by the runtime system.

The `-mblock-move-inline-limit` command line option has been added, which enables the user to control the maximum size of inlined `memcpy` calls and similar.

4.1.1.2.1.5. Link-time Optimization

Link-time optimization (LTO) is a compilation technique in which GCC generates an internal representation of each compiled input file in addition to the native code, and writes both to the output object file.

Subsequently, when several object files are linked together, GCC uses the internal representations of the compiled code to optimize inter-procedurally across all the compilation units. This can potentially improve the performance of the generated code (for example, functions defined in one file can potentially be inlined when called in another file).

To enable LTO, the `-flto` option needs to be specified at both compile time and link time. For further details, including interoperability with linkers and parallel execution of LTO, refer to the documentation for `-flto` in the [GCC 4.7.0 Manual](#). Also note that the internal representation is not a stable interface, so LTO will only apply to code generated by the same version of GCC.



Note

Use of Link-time Optimization with debug generation is not yet supported in gcc 4.7 and 4.8 and so use of the `-flto` and the `-g` options together is unsupported in Red Hat Developer Toolset.

4.1.1.2.1.6. Miscellaneous

`-Ofast` is now supported as a general optimization level. It operates similar to `-O3`, adds options that can yield better-optimized code, but in turn might invalidate standards compliance (for example, `-ffast-math` is enabled by `-Ofast`).

GCC can now inform users about cases in which code generation might be improved by adding attributes such as `const`, `pure`, and `noreturn` to functions declared in header files. Use the `-Wsuggest-attribute=[const|pure|noreturn]` command line option to enable this.

Assembler code can now make use of a `goto` feature that allows for jumps to labels in C code.

4.1.1.2.2. Language Compatibility

In this section, we describe the compatibility between the Red Hat Developer Toolset compilers and the Red Hat Enterprise Linux system compilers at the programming-language level (for example, differences in the implementation of language standards such as C99, or changes to the warnings generated by `-Wall`).

Some of the changes are a result of bug fixing, and some old behaviors have been intentionally changed in order to support new standards, or relaxed in standards-conforming ways to facilitate compilation or runtime performance. Some of these changes are not visible to the naked eye and will not cause problems when updating from older versions. However, some of these changes are visible, and can cause grief to users porting to Red Hat Developer Toolset's version of GCC. The following text attempts to identify major issues and suggests solutions.

4.1.1.2.2.1. C

Constant expressions are now handled by GCC in a way that conforms to C90 and C99. For code expressions that can be transformed into constants by the compiler but are in fact not constant expressions as defined by ISO C, this may cause warnings or errors.

Ill-formed redeclarations of library functions are no longer accepted by the compiler. In particular, a function with a signature similar to the built-in declaration of a library function (for example, `abort()` or `memcpy()`) must be declared with `extern "C"` to be considered as a redeclaration, otherwise it is ill-formed.

Duplicate Member

Consider the following **struct** declaration:

```
struct A { int *a; union { struct { int *a; }; }; };
```

Previously, this declaration used to be diagnosed just by the C++ compiler, now it is also diagnosed by the C compiler. Because of the anonymous unions and structs, there is ambiguity about what `.a` actually refers to and one of the fields therefore needs to be renamed.

4.1.1.2.2.2. C++

Header Dependency Changes

`<iostream>`, `<string>`, and other STL headers that previously included `<unistd.h>` as an implementation detail (to get some feature macros for `gthr*.h` purposes) no longer do so, because it was a C++ standard violation. This can result in diagnostic output similar to the following:

```
error: 'truncate' was not declared in this scope
error: 'sleep' was not declared in this scope
error: 'pipe' was not declared in this scope
error: there are no arguments to 'offsetof' that depend on a template
parameter, so a declaration of 'offsetof' must be available
```

To fix this, add the following line early in the source or header files that need it:

```
#include <unistd.h>
```

Many of the standard C++ library include files have been edited to no longer include `<cstddef>` to get namespace-`std`-scoped versions of `size_t` and `ptrdiff_t`. As such, C++ programs that used the macros `NULL` or `offsetof` without including `<cstddef>` will no longer compile. The diagnostic produced is similar to the following:

```
error: 'ptrdiff_t' does not name a type
error: 'size_t' has not been declared
error: 'NULL' was not declared in this scope
error: there are no arguments to 'offsetof' that depend on a template
parameter, so a declaration of 'offsetof' must be available
```

To fix this issue, add the following line:

```
#include <cstddef>
```

Name Lookup Changes

G++ no longer performs an extra unqualified lookup that it incorrectly performed in the past. Instead, it implements the two-phase lookup rules correctly, and an unqualified name used in a template must have an appropriate declaration that:

1. is either in scope at the point of the template's definition, or
2. can be found by argument-dependent lookup at the point of instantiation.

Code that incorrectly depends on a second unqualified lookup at the point of instantiation (such as finding functions declared after the template or in dependent bases) will result in compile-time errors.

In some cases, the diagnostics provided by G++ include hints how to fix the bugs. Consider the following

code:

```
template<typename T>
int t(T i)
{
    return f(i);
}

int f(int i)
{
    return i;
}

int main()
{
    return t(1);
}
```

The following diagnostics output will be produced:

```
In instantiation of 'int t(T) [with T = int]'
required from here
error: 'f' was not declared in this scope, and no declarations were found by
argument-dependent lookup at the point of instantiation [-fpermissive]
note: 'int f(int)' declared here, later in the translation unit
```

To correct the error in this example, move the declaration of function **f()** before the definition of template function **t()**. The **-fpermissive** compiler flag turns compile-time errors into warnings and can be used as a temporary workaround.

Uninitialized const

Consider the following declaration:

```
struct A { int a; A (); };
struct B : public A { };
const B b;
```

An attempt to compile this code now fails with the following error:

```
error: uninitialized const 'b' [-fpermissive]
note: 'const struct B' has no user-provided default constructor
```

This happens, because **B** does not have a user-provided default constructor. Either an initializer needs to be provided, or the default constructor needs to be added.

Visibility of Template Instantiations

The ELF symbol visibility of a template instantiation is now properly constrained by the visibility of its template arguments. For instance, users that instantiate standard library components like **std::vector** with hidden user defined types such as **struct my_hidden_struct** can now expect hidden visibility for **std::vector<my_hidden_struct>** symbols. As a result, users that compile with the **-fvisibility=hidden** command line option should be aware of the visibility of types included from the library headers used. If the header does not explicitly control symbol visibility, types from those headers will be hidden, along with instantiations that use those types. For instance, consider the following code:

```
#include <vector>           // template std::vector has default visibility
#include <ctime>            // struct tm has hidden visibility
template class std::vector<tm>; // instantiation has hidden visibility
```

One approach to adjusting the visibility of a library header `<foo.h>` is to create a forwarding header on the `-I` include path consisting of the following:

```
#pragma GCC visibility push(default)
#include_next <foo.h>
#pragma GCC visibility push
```

User-defined Literal Support

When compiling C++ with the `-std={c++11,c++0x,gnu++11,gnu++0x}` command line option, GCC 4.7.0 and later, unlike older versions, supports user-defined literals, which are incompatible with some valid ISO C++03 code. In particular, white space is now needed after a string literal before something that could be a valid user defined literal. Consider the following code:

```
const char *p = "foobar"__TIME__;
```

In C++03, the `__TIME__` macro expands to some string literal and is concatenated with the other one. In C++11, `__TIME__` is not expanded and instead, operator `" __TIME__` is being looked up, which results in a warning like:

```
error: unable to find string literal operator 'operator" __TIME__'
```

This applies to any string literal followed without white space by some macro. To fix this, add some white space between the string literal and the macro name.

Taking the Address of Temporary

Consider the following code:

```
struct S { S (); int i; };
void bar (S *);
void foo () { bar (&S ()); }
```

Previously, an attempt to compile this code produced a warning message, now it fails with an error. This can be fixed by adding a variable and passing the address of this variable instead of the temporary. The `-fpermissive` compiler flag turns compile-time errors into warnings and can be used as a temporary workaround.

Miscellaneous

G++ now sets the predefined macro `__cplusplus` to the correct value: `199711L` for C++98/03, and `201103L` for C++11.

G++ now properly re-uses stack space allocated for temporary objects when their lifetime ends, which can significantly lower stack consumption for some C++ functions. As a result of this, some code with undefined behavior will now break.

When an extern declaration within a function does not match a declaration in the enclosing context, G++ now properly declares the name within the namespace of the function rather than the namespace which was open just before the function definition.

G++ now implements the proposed resolution of the C++ standard's core issue 253. Default initialization is allowed if it initializes all subobjects, and code that fails to compile can be fixed by providing an initializer such as:

```
struct A { A(); };
struct B : A { int i; };
const B b = B();
```

Access control is now applied to **typedef** names used in a template, which may cause G++ to reject some ill-formed code that was accepted by earlier releases. The **-fno-access-control** option can be used as a temporary workaround until the code is corrected.

G++ now implements the C++ standard's core issue 176. Previously, G++ did not support using the injected-class-name of a template base class as a type name, and lookup of the name found the declaration of the template in the enclosing scope. Now lookup of the name finds the injected-class-name, which can be used either as a type or as a template, depending on whether or not the name is followed by a template argument list. As a result of this change, some code that was previously accepted may be ill-formed, because:

1. the injected-class-name is not accessible because it is from a private base, or
2. the injected-class-name cannot be used as an argument for a template parameter.

In either of these cases, the code can be fixed by adding a nested-name-specifier to explicitly name the template. The first can be worked around with **-fno-access-control**, the second is only rejected with **-pedantic**.

4.1.1.2.2.3. C/C++ Warnings

GCC 4.7.0 and later adds a number of new warnings that are either enabled by default, or by using the **-Wall** option. Although these warnings do not result in a compilation failure on their own, often **-Wall** is used in conjunction with **-Werror**, causing these warnings to act like errors. This section provides a list of these new or newly enabled warnings. Unless noted otherwise, these warnings apply to both C and C++.

The behavior of the **-Wall** command line option has changed and now includes the new warning flags **-Wunused-but-set-variable** and, with **-Wall -Wextra**, **-Wunused-but-set-parameter**. This may result in new warnings in code that compiled cleanly with previous versions of GCC. For example, consider the following code:

```
void fn (void)
{
    int foo;
    foo = bar (); /* foo is never used. */
}
```

The following diagnostic output will be produced:

```
warning: variable "foo" set but not used [-Wunused-but-set-variable]
```

To fix this issue, first see if the unused variable or parameter can be removed without changing the result or logic of the surrounding code. If not, annotate it with **__attribute__((__unused__))**. As a workaround, you can use the **-Wno-error=unused-but-set-variable** or **-Wno-error=unused-but-set-parameter** command line option.

The **-Wenum-compare** option causes GCC to report a warning when values of different enum types are being compared. Previously, this option only worked for C++ programs, but now it works for C as well. This warning is enabled by **-Wall** and may be avoided by using a type cast.

Casting integers to larger pointer types now causes GCC to display a warning by default. To disable these warnings, use the **-Wno-int-to-pointer-cast** option, which is available for both C and C++.

Conversions between NULL and non-pointer types now cause GCC to report a warning by default. Previously, these warnings were only displayed when explicitly using **-Wconversion**. To disable these warnings, use the new **-Wno-conversion-null** command line option.

GCC can now warn when a class that has virtual functions and a non-virtual destructor is destroyed by using **delete**. This is unsafe to do because the pointer might refer to a base class that does not have a virtual destructor. The warning is enabled by **-Wall** and by a new command line option, **-Wdelete-non-virtual-dtor**.

New **-Wc++11-compat** and **-Wc++0x-compat** options are now available. These options cause GCC to display a warning about C++ constructs whose meaning differs between ISO C++ 1998 and ISO C++ 2011 (such as identifiers in ISO C++ 1998 that are keywords in ISO C++ 2011). This warning is enabled by **-Wall** and enables the **-Wnarrowing** option.

4.1.1.2.2.4. Fortran

4.1.1.2.2.4.1. New Features

- ▶ A new compile flag **-fstack-arrays** has been added. This flag causes all local arrays to be put on stack memory, which can significantly improve the performance of some programs. Note that programs that use very large local arrays may require you to extend your runtime limits for stack memory.
- ▶ Compile time has been significantly improved. For example, the improvement may be noticeable when working with programs that use large array constructors.
- ▶ To improve code generation and diagnostics, the **-fwhole-file** compile flag is now enabled by default, and can be used with a newly supported **-fwhole-program** flag. To disable it, use the deprecated **-fno-whole-file** flag.
- ▶ A new command line option **-M** is now supported. Similarly to **gcc**, this option allows you to generate Makefile dependencies. Note that the **-cpp** option may be required as well.
- ▶ The **-finit-real=** command line option now supports **snan** as a valid value. This allows you to initialize REAL and COMPLEX variables with a signaling NaN (*not a number*), and requires you to enable trapping (for example, by using the **-ffpe-trap=** command line option). Note that compile-time optimizations may turn a signaling NaN into a quiet NaN.
- ▶ A new command line option **-fcheck=** has been added. This option accepts the following arguments:
 - The **-fcheck=bounds** option is equivalent to the **-fbounds-check** command line option.
 - The **-fcheck=array-temps** option is equivalent to the **-fcheck-array-temporaries** command line option.
 - The **-fcheck=do** option checks for invalid modification of loop iteration variables.
 - The **-fcheck=recursive** option checks for recursive calls to subroutines or functions that are not marked as recursive.

- The **-fcheck=pointer** option performs pointer association checks in calls, but does not handle undefined pointers nor pointers in expressions.
- The **-fcheck=all** option enables all of the above options.
- ▶ A new command line option **-fno-protect-parens** has been added. This option allows the compiler to reorder REAL and COMPLEX expressions with no regard to parentheses.
- ▶ When OpenMP's **WORKSHARE** is used, array assignments and **WHERE** will now be run in parallel.
- ▶ More Fortran 2003 and Fortran 2008 mathematical functions can now be used as initialization expressions.
- ▶ The **GCC\$** compiler directive now enables support for some extended attributes such as **STDCALL**.

4.1.1.2.2.4.2. Compatibility Changes

- ▶ The **-Ofast** command line option now automatically enables the **-fno-protect-parens** and **-fstack-arrays** flags.
- ▶ Front-end optimizations can now be disabled by the **-fno-frontend-optimize** option, and selected by the **-ffrontend-optimize** option. The former is essentially only desirable if invalid Fortran source code needs to be compiled (for example, when functions—as compared to subroutines—have side-effects) or to work around compiler bugs.
- ▶ The **GFORTTRAN_USE_STDERR** environment variable has been removed, and GNU Fortran now always prints error messages to standard error.
- ▶ The **-fdump-core** command line option and the **GFORTTRAN_ERROR_DUMPCORE** environment variable have been removed. When encountering a serious error, GNU Fortran now always aborts the execution of the program.
- ▶ The **-fbacktrace** command line option is now enabled by default. When a fatal error occurs, GNU Fortran now attempts to print a backtrace to standard error before aborting the execution of the program. To disable this behavior, use the **-fno-backtrace** option.
- ▶ GNU Fortran no longer supports the use of the **-M** command line option to generate Makefile dependencies for the module path. To perform this operation, use the **-J** option instead.
- ▶ To significantly reduce the number of warnings, the **-Wconversion** command line option now only displays warnings when a conversion leads to information loss, and a new command line option **-Wconversion-extra** has been added to display warnings about other conversions. The **-Wconversion** option is now enabled with **-Wall**.
- ▶ A new command line option **-Wunused-dummy-argument** has been added. This option can be used to display warnings about unused dummy arguments, and is now enabled with **-Wall**. Note that the **-Wunused-variable** option previously also warned about unused dummy arguments.
- ▶ The **COMMON** default padding has been changed. Previously, the padding was added before a variable. Now it is added after a variable to increase the compatibility with other vendors, as well as to help to obtain the correct output in some cases. Note that this behavior is in contrast with the behavior of the **-falign-commons** option.

- ▶ GNU Fortran no longer links against the **libgfortranbegin** library. The **MAIN__** assembler symbol is the actual Fortran main program and is invoked by the **main** function, which is now generated and put in the same object file as **MAIN__**. Note that the **libgfortranbegin** library is still present for backward compatibility.

4.1.1.2.2.4.3. Fortran 2003 Features

- ▶ Improved but still experimental support for polymorphism between libraries and programs and for complicated inheritance patterns.
- ▶ Generic interface names which have the same name as derived types are now supported, which allows the creation of constructor functions. Note that Fortran does not support static constructor functions; only default initialization or an explicit structure-constructor initialization are available.
- ▶ Automatic (re)allocation: In intrinsic assignments to allocatable variables, the left-hand side will be automatically allocated (if unallocated) or reallocated (if the shape or type parameter is different). To avoid the small performance penalty, you can use **a(:) = ...** instead of **a = ...** for arrays and character strings — or disable the feature using **-std=f95** or **-fno-realloc-lhs**.
- ▶ Experimental support of the **ASSOCIATE** construct has been added.
- ▶ In pointer assignments it is now possible to specify the lower bounds of the pointer and, for a rank-1 or a simply contiguous data-target, to remap the bounds.
- ▶ Deferred type parameter: For scalar allocatable and pointer variables the character length can now be deferred.
- ▶ Namelist variables with allocatable attribute, pointer attribute, and with a non-constant length type parameter are now supported.
- ▶ Support has been added for procedure-pointer function results and procedure-pointer components (including **PASS**).
- ▶ Support has been added for allocatable scalars (experimental), **DEFERRED** type-bound procedures, and the **ERRMSG=** argument of the **ALLOCATE** and **DEALLOCATE** statements.
- ▶ The **ALLOCATE** statement now supports type-specs and the **SOURCE=** argument.
- ▶ Rounding (**ROUND=**, **RZ**, ...) for output is now supported.
- ▶ The **INT_FAST{8,16,32,64,128}_T** format for **ISO_C_BINDING** intrinsic module type parameters is now supported.
- ▶ **OPERATOR(*)** and **ASSIGNMENT(=)** are now allowed as **GENERIC** type-bound procedures (i.e. as type-bound operators).

4.1.1.2.2.4.4. Fortran 2003 Compatibility

Extensible derived types with type-bound procedure or procedure pointer with **PASS** attribute now have to use **CLASS** in line with the Fortran 2003 standard; the workaround to use **TYPE** is no longer supported.

4.1.1.2.2.4.5. Fortran 2008 Features

- ▶ A new command line option **-std=f2008ts** has been added. This option enables support for programs that conform to the Fortran 2008 standard and the draft Technical Specification (TS) 29113 on Further Interoperability of Fortran with C. For more information, refer to the [Chart of Fortran TS 29113 Features supported by GNU Fortran](#).
- ▶ The **DO CONCURRENT** construct is now supported. This construct can be used to specify that individual loop iterations do not have any interdependencies.
- ▶ Full single-image support except for polymorphic coarrays has been added, and can be enabled by using the **-fcoarray=single** command line option. Additionally, GNU Fortran now provides preliminary support for multiple images via an MPI-based coarray communication library. Note that the library version is not yet usable as remote coarray access is not yet possible.
- ▶ The **STOP** and **ERROR STOP** statements have been updated to support all constant expressions.
- ▶ The **CONTIGUOUS** attribute is now supported.
- ▶ Use of **ALLOCATE** with the **MOLD** argument is now supported.
- ▶ The **STORAGE_SIZE** intrinsic inquiry function is now supported.
- ▶ The **NORM2** and **PARITY** intrinsic functions are now supported.
- ▶ The following bit intrinsics have been added:
 - the **POPCNT** and **POPPAR** bit intrinsics for counting the number of 1 bits and returning the parity;
 - the **BGE**, **BGT**, **BLE**, and **BLT** bit intrinsics for bitwise comparisons;
 - the **DSHIFTL** and **DSHIFTR** bit intrinsics for combined left and right shifts;
 - the **MASKL** and **MASKR** bit intrinsics for simple left and right justified masks;
 - the **MERGE_BITS** bit intrinsic for a bitwise merge using a mask;
 - the **SHIFTA**, **SHIFTL**, and **SHIFTR** bit intrinsics for shift operations;
 - the transformational bit intrinsics **IALL**, **IANY**, and **IPARITY**.
- ▶ The **EXECUTE_COMMAND_LINE** intrinsic subroutine is now supported.
- ▶ The **IMPURE** attribute for procedures is now supported. This allows the use of **ELEMENTAL** procedures without the restrictions of **PURE**.
- ▶ Null pointers (including **NULL()**) and unallocated variables can now be used as an actual argument to optional non-pointer, non-allocatable dummy arguments, denoting an absent argument.
- ▶ Non-pointer variables with the **TARGET** attribute can now be used as an actual argument to **POINTER** dummies with **INTENT (IN)**.
- ▶ Pointers that include procedure pointers and those in a derived type (pointer components) can now also be initialized by a target instead of only by **NULL**.
- ▶ The **EXIT** statement (with construct-name) can now be used to leave the **ASSOCIATE**, **BLOCK**, **IF**, **SELECT CASE**, and **SELECT TYPE** constructs in addition to **DO**.
- ▶ Internal procedures can now be used as actual arguments.

- ▶ The named constants **INTEGER_KINDS**, **LOGICAL_KINDS**, **REAL_KINDS**, and **CHARACTER_KINDS** of the intrinsic module **ISO_FORTRAN_ENV** have been added. These arrays contain the supported 'kind' values for the respective types.
- ▶ The **C_SIZEOF** module procedures of the **ISO_C_BINDINGS** intrinsic module and the **COMPILER_VERSION** and **COMPILER_OPTIONS** module procedures of the **ISO_FORTRAN_ENV** intrinsic module have been implemented.
- ▶ The **OPEN** statement now supports the **NEWUNIT=** option. This option returns a unique file unit and therefore prevents inadvertent use of the same unit in different parts of the program.
- ▶ Unlimited format items are now supported.
- ▶ The **INT{8,16,32}** and **REAL{32,64,128}** format for **ISO_FORTRAN_ENV** intrinsic module type parameters are now supported.
- ▶ It is now possible to use complex arguments with the **TAN**, **SINH**, **COSH**, **TANH**, **ASIN**, **ACOS**, and **ATAN** functions. Additionally, the new functions **ASINH**, **ACOSH**, and **ATANH** have been added for real and complex arguments, and **ATAN(Y,X)** now serves as an alias for **ATAN2(Y,X)**.
- ▶ The **BLOCK** construct has been implemented.

4.1.1.2.2.4.6. Fortran 2008 Compatibility

The implementation of the **ASYNCHRONOUS** attribute in GCC is now compatible with the candidate draft of *TS 29113: Technical Specification on Further Interoperability with C*.

4.1.1.2.2.4.7. Fortran 77 Compatibility

When the GNU Fortran compiler is issued with the **-fno-sign-zero** option, the **SIGN** intrinsic now behaves as if zero were always positive.

4.1.1.2.3. ABI Compatibility

This section describes compatibility between the Red Hat Developer Toolset compilers and the system compilers at the *application binary interface* (ABI) level.

4.1.1.2.3.1. C++ ABI

Because the upstream GCC community development does not guarantee C++11 ABI compatibility across major versions of GCC, the same applies to use of C++11 with Red Hat Developer Toolset. Consequently, using the **-std=c++11** option is supported in Red Hat Developer Toolset 2.0 only when all C++ objects compiled with that flag have been built using the same major version of Red Hat Developer Toolset. The mixing of objects, binaries and libraries, built by the Red Hat Enterprise Linux 5 or 6 system toolchain GCC using the **-std=c++0x** or **-std=gnu++0x** flags, with those built with the **-std=c++11** or **-std=gnu++11** flags using the GCC in Red Hat Developer Toolset is explicitly not supported.

As later major versions of Red Hat Developer Toolset may use a later major release of GCC, forward-compatibility of objects, binaries, and libraries built with the **-std=c++11** or **-std=gnu++11** options cannot be guaranteed, and so is not supported.

The default language standard setting for Red Hat Developer Toolset is C++98. Any C++98-compliant binaries or libraries built in this default mode (or explicitly with **-std=c++98**) can be freely mixed with binaries and shared libraries built by the Red Hat Enterprise Linux 5 or 6 system toolchain GCC. Red Hat recommends use of this default **-std=c++98** mode for production software development.



Important

Use of C++11 features in your application requires careful consideration of the above ABI compatibility information.

Aside from the C++11 ABI, discussed above, [the Red Hat Enterprise Linux Application Compatibility Specification](#) is unchanged for Red Hat Developer Toolset. When mixing objects built with Red Hat Developer Toolset with those built with the Red Hat Enterprise Linux v5.x/v6.x toolchain (particularly `.o/a` files), the Red Hat Developer Toolset toolchain should be used for any linkage. This ensures any newer library features provided only by Red Hat Developer Toolset are resolved at link-time.

A new standard mangling for SIMD vector types has been added to avoid name clashes on systems with vectors of varying length. By default the compiler still uses the old mangling, but emits aliases with the new mangling on targets that support strong aliases. `-Wabi` will now display a warning about code that uses the old mangling.

4.1.1.2.3.2. Miscellaneous

GCC now optimizes calls to various standard C string functions such as `strlen()`, `strchr()`, `strcpy()`, `strcat()` and `stpcpy()` (as well as their respective `_FORTIFY_SOURCE` variants) by transforming them into custom, faster code. This means that there might be fewer or other calls to those functions than in the original source code. The optimization is enabled by default at `-O2` or higher optimization levels. It is disabled when using `-fno-optimize-strlen` or when optimizing for size.

When compiling for 32-bit GNU/Linux and not optimizing for size, `-fomit-frame-pointer` is now enabled by default. The prior default setting can be chosen by using the `-fno-omit-frame-pointer` command line option.

Floating-point calculations on x86 targets and in strict C99 mode are now compiled by GCC with a stricter standard conformance. This might result in those calculations executing significantly slower. It can be disabled using `-fexcess-precision=fast`.

4.1.1.2.4. Debugging Compatibility

GCC now generates DWARF debugging information that uses more or newer DWARF features than previously. GDB contained in Red Hat Developer Toolset can handle these features, but versions of GDB older than 7.0 cannot. GCC can be restricted to only generate debugging information with older DWARF features by using the `-gdwarf-2 -gstrict-dwarf` or `-gdwarf-3 -gstrict-dwarf` options (the latter are handled partially by versions of GDB older than 7.0).

Many tools such as `Valgrind`, `SystemTap`, or third-party debuggers utilize debugging information. It is suggested to use the `-gdwarf-2 -gstrict-dwarf` options with those tools.



Note

Use of Link-time Optimization with debug generation is not yet supported in gcc 4.7 and 4.8 and so use of the `-flto` and the `-g` options together is unsupported in Red Hat Developer Toolset.

4.1.1.2.5. Other Compatibility

GCC is now more strict when parsing command line options, and both **gcc** and **g++** report an error when invalid command line options are used. In particular, when only linking and not compiling code, earlier versions of GCC ignored all options starting with `--`. For example, options accepted by the linker such as `--as-needed` and `--export-dynamic` are not accepted by **gcc** and **g++** anymore, and should now be directed to the linker using `-Wl,--as-needed` or `-Wl,--export-dynamic` if that is intended.

Because of the new link-time optimization feature (see [Section 4.1.1.2.1.5, “Link-time Optimization”](#)), support for the older intermodule optimization framework has been removed and the `-combine` command line option is not accepted anymore.

4.2. Distributed Compiling

Red Hat Enterprise Linux also supports *distributed compiling*. This involves transforming one compile job into many smaller jobs; these jobs are distributed over a cluster of machines, which speeds up build time (particularly for programs with large codebases). The **distcc** package provides this capability.

To set up distributed compiling, install the following packages:

- ▶ **distcc**
- ▶ **distcc-server**

For more information about distributed compiling, refer to the **man** pages for **distcc** and **distccd**. The following link also provides detailed information about the development of **distcc**:

<http://code.google.com/p/distcc>

4.3. Autotools

GNU Autotools is a suite of command line tools that allow developers to build applications on different systems, regardless of the installed packages or even Linux distribution. These tools aid developers in creating a **configure** script. This script runs prior to builds and creates the top-level **Makefiles** required to build the application. The **configure** script may perform tests on the current system, create additional files, or run other directives as per parameters provided by the builder.

The Autotools suite's most commonly-used tools are:

autoconf

Generates the **configure** script from an input file (**configure.ac**, for example)

automake

Creates the **Makefile** for a project on a specific system

autoscan

Generates a preliminary input file (that is, **configure.scan**), which can be edited to create a final **configure.ac** to be used by **autoconf**

All tools in the Autotools suite are part of the **Development Tools** group package. You can install this package group to install the entire Autotools suite, or use **yum** to install any tools in the suite as you wish.

4.3.1. Autotools Plug-in for Eclipse

The Autotools suite is also integrated into the Eclipse IDE via the Autotools plug-in. This plug-in provides an Eclipse graphical user interface for Autotools, which is suitable for most C/C++ projects.

As of Red Hat Enterprise Linux 6, this plug-in only supports two templates for new C/C++ projects:

- ▶ An empty project
- ▶ A "hello world" application

The empty project template is used when importing projects into the C/C++ Development Toolkit that already support Autotools. Future updates to the Autotools plug-in will include additional graphical user interfaces (wizards, for example) for creating shared libraries and other complex scenarios.

The Red Hat Enterprise Linux 6 version of the Autotools plug-in also does not integrate **git** or **mercurial** into Eclipse. As such, Autotools projects that use **git** repositories will be required to be checked out *outside* the Eclipse workspace. Afterwards, you can specify the source location for such projects in Eclipse. Any repository manipulation (commits, or updates for example) are done via the command line.

4.3.2. Configuration Script

The most crucial function of Autotools is the creation of the **configure** script. This script tests systems for tools, input files, and other features it can use in order to build the project [2]. The **configure** script generates a **Makefile** which allows the **make** tool to build the project based on the system configuration.

To create the **configure** script, first create an input file. Then feed it to an Autotools utility in order to create the **configure** script. This input file is typically **configure.ac** or **Makefile.am**; the former is usually processed by **autoconf**, while the later is fed to **automake**.

If a **Makefile.am** input file is available, the **automake** utility creates a **Makefile** template (that is, **Makefile.in**), which may refer to information collected at configuration time. For example, the **Makefile** may have to link to a particular library *if and only if* that library is already installed. When the **configure** script runs, **automake** will use the **Makefile.in** templates to create a **Makefile**.

If a **configure.ac** file is available instead, then **autoconf** will automatically create the **configure** script based on the macros invoked by **configure.ac**. To create a preliminary **configure.ac**, use the **autoscan** utility and edit the file accordingly.

4.3.3. Autotools Documentation

Red Hat Enterprise Linux includes **man** pages for **autoconf**, **automake**, **autoscan** and most tools included in the Autotools suite. In addition, the Autotools community provides extensive documentation on **autoconf** and **automake** on the following websites:

- ▶ <http://www.gnu.org/software/autoconf/manual/autoconf.html>
- ▶ <http://www.gnu.org/software/autoconf/manual/automake.html>

The following is an online book describing the use of Autotools. Although the above online documentation is the recommended and most up to date information on Autotools, this book is a good alternative and introduction.

- ▶ <http://sourceware.org/autobook/>

For information on how to create Autotools input files, refer to:

- ▶ <http://www.gnu.org/software/autoconf/manual/autoconf.html#Making-configure-Scripts>
- ▶ <http://www.gnu.org/software/autoconf/manual/automake.html#Invoking-Automake>

The following upstream example also illustrates the use of Autotools in a simple **hello** program:

- ▶ <http://www.gnu.org/software/hello/manual/hello.html>

The *Autotools Plug-in For Eclipse* whitepaper also provides more detail on the Red Hat Enterprise Linux 6 release of the Autotools plug-in. This whitepaper also includes a "by example" case study to walk you through a typical use-case for the plug-in. Refer to the following link for more information:

https://access.redhat.com/sites/default/files/red_hat_enterprise_linux-6-autotools_plug-in_for_eclipse-en-us.pdf

4.4. Eclipse Built-in Specfile Editor

The Specfile Editor Plug-in for Eclipse provides useful features to help developers manage **.spec** files. This plug-in allows users to leverage several Eclipse GUI features in editing **.spec** files, such as auto-completion, highlighting, file hyperlinks, and folding.

In addition, the Specfile Editor Plug-in also integrates the **rpmlint** tool into the Eclipse interface. **rpmlint** is a command line tool that helps developers detect common RPM package errors. The richer visualization offered by the Eclipse interface helps developers quickly detect, view, and correct mistakes reported by **rpmlint**.

The Specfile Editor for Eclipse is provided by the **eclipse-rpm-editor** package. For more information about this plug-in, refer to *Specfile Editor User Guide* in the Eclipse **Help Contents**.

4.5. CDT in Eclipse

The *CDT* (C/C++ Development Tools) is an Eclipse project that adds support for developing C and C++ projects with Eclipse. A user can create three forms of projects:

1. Managed Make Project
2. Standard Make Project
3. Autotools Project

4.5.1. Managed Make Project

A managed make CDT project, sometimes referred to as a managed project, is one where the details regarding how to build the project are automated on behalf of the end-user. This differs from the standard make project, another common type of CDT C/C++ project, whereby a user supplies a Makefile that has the build details specified.

For a managed project, start by selecting the type of managed project and the required toolchain. The project type is categorized based on the ultimate target of the project, such as an executable, a shared library, or a static library. Within these categories there may be templates for more specific projects (a hello world sample executable project, for example) where base source files are already provided. These can then be further customized.

A toolchain is the set of tools used to generate the target. Typically, a Red Hat Enterprise Linux C/C++ developer would select the Linux GCC toolchain, which uses GCC for compiling, linking and assembly. Each tool in the toolchain is associated with one or more input types, typically specified by file suffix (for example, `.c` or `.h` or `.S`) or by file name. The tool will have parameter settings which can be customized by the developer, and each tool has an output type it creates. The tool also has a command or binary executable associated with it which may overlap among multiple tools. The C compiler and linker, for example, can both use GCC, but the compiler and linker tools will each have different input and output types, as well as different settings presented to the developer. Customize the tool settings through **Properties > C/C++ Build > Settings**. The toolchain itself can be customized in order to add, remove, or replace the used tools through **Properties > C/C++ Build > Toolchain Editor**.

New files, such as source files or header files, can be added to the project once it has been created. The new files are added automatically to the build based on their input types and the tool settings. Navigate to **Project > C/C++ Build** under the **Builder Settings** tab in order for the managed make project to generate a Makefile which can be distributed with the project. This facilitates the use of the Makefile outside Eclipse.

For more information regarding managed make C/C++ projects, refer to the *C/C++ Development User Guide* which can be found by navigating to **Concepts > Project Types, Tasks > Creating a Project**, or **Reference > C/C++ Properties > C/C++ Project Properties > C/C++ Build > Settings Page**.

4.5.2. Standard Make Project

A standard make CDT project is a traditional C project with a Makefile that is manually managed by the developer. Unlike the managed make project, there are no tool settings to be used in determining rules in the Makefile. Manual additions to the Makefile are required when new source files that are to be processed as part of the build are added to the project. If a pattern rule exists which the new file name matches (for example, `.c : .o`, which states how to process a file with the suffix `.c` into a file with the suffix `.o`), then this is not required.

The default make target for building the project is **all**, and the default make target for cleaning the project is **clean**. It is also possible for a user to build other targets found in the Makefile. To do this use the Makefile Target dialog to create targets to run or build existing ones. Creating pseudo-targets that group multiple targets found in the makefile in a particular order is also done through the Makefile Target dialog. To access the specific create and build dialogs, select **Project > Make Target > Create...** and **Project > Make Target > Build...** respectively. Alternatively, right click on **resources** in the project and select the **Make Targets** option to access either **Create...** or **Build...**

For more information regarding standard make C/C++ projects, refer to the *C/C++ Development user guide*, accessed through **Concepts > Project Types, Tasks > Creating a project** or **Reference > C/C++ Properties > C/C++ Project Properties > C/C++Build > Settings Page**.

4.5.3. Autotools Project

An autotools project is a lot like a standard make project however the Makefile is usually generated as part of a configuration step that occurs prior to building. See [Section 4.3, "Autotools"](#) for details on Autotools and the Autotools plug-in for Eclipse which adds support for this type of project. Like a standard make project, make targets can be run with the Make Target dialog.

4.6. build-id Unique Identification of Binaries

Each executable or shared library built with Red Hat Enterprise Linux Server 6 or later is assigned a unique identification 160-bit SHA-1 string, generated as a checksum of selected parts of the binary. This allows two builds of the same program on the same host to always produce consistent build-ids and binary content.

Display the build-id of a binary with the following command:

```
$ eu-readelf -n usr/bin/bash
[...]
Note section [ 3] '.note.gnu.build-id' of 36 bytes at offset 0x274:
  Owner   Data size   Type
  GNU     20        GNU_BUILD_ID
  Build ID: efdd0b5e69b0742fa5e5bad0771df4d1df2459d1
```

Unique identifiers of binaries are useful in cases such as analysing core files, documented [Section 5.2.1, “Installing Debuginfo Packages for Core Files Analysis”](#).

4.7. Software Collections and `scl-utils`

With Software Collections, it is possible to build and concurrently install multiple versions of the same RPM packages on a system. Software Collections have no impact on the system versions of the packages installed by the conventional RPM package manager.

To enable support for Software Collections on a system, install the packages `scl-utils` and `scl-utils-build` by typing the following at a shell prompt:

```
# yum install -y scl-utils scl-utils-build
```

The `scl-utils` package provides the `scl` tool which is used to enable a Software Collection and to run applications in the Software Collection environment.

General usage of the `scl` tool can be described using the following syntax:

```
scl action software_collection_1 software_collection_2 command
```

Example 4.1. Running an Application Directly

To directly run **Perl** with the `--version` option in the Software Collection named `software_collection_1`, execute the following command:

```
scl enable software_collection_1 'perl --version'
```

Example 4.2. Running a Shell with Multiple Software Collections Enabled

To run the **Bash** shell in the environment with multiple Software Collections enabled, execute the following command:

```
scl enable software_collection_1 software_collection_2 bash
```

The command above enables two Software Collections named `software_collection_1` and `software_collection_2`.

Example 4.3. Running Commands Stored in a File

To execute a number of commands, which are stored in a file, in the Software Collection environment, run the following command:

```
cat cmd | scl enable software_collection_1 -
```

The command above executes commands, which are stored in the **cmd** file, in the environment of the Software Collection named **software_collection_1**.

For more information regarding Software Collections and **scl-utils**, refer to the *Software Collections Guide* as part of the *Red Hat Developer Toolset*.

[2] For information about tests that **configure** can perform, refer to the following link:

<http://www.gnu.org/software/autconf/manual/autconf.html#Existing-Tests>

Chapter 5. Debugging

Useful, well-written software generally goes through several different phases of application development, allowing ample opportunity for mistakes to be made. Some phases come with their own set of mechanisms to detect errors. For example, during compilation an elementary semantic analysis is often performed to make sure objects, such as variables and functions, are adequately described.

The error-checking mechanisms performed during each application development phase aims to catch simple and obvious mistakes in code. The debugging phase helps to bring more subtle errors to light that fell through the cracks during routine code inspection.

5.1. ELF Executable Binaries

Red Hat Enterprise Linux uses ELF for executable binaries, shared libraries, or debuginfo files. Within these debuginfo ELF files, the DWARF format is used. Version 3 of DWARF is used in ELF files (that is, `gcc -g` is equivalent to `gcc -gdwarf-3`). DWARF debuginfo includes:

- ▶ names of all the compiled functions and variables, including their target addresses in binaries
- ▶ source files used for compilation, including their source line numbers
- ▶ local variables location



Important

STABS is occasionally used with UNIX. STABS is an older, less capable format. Its use is discouraged by Red Hat. GCC and GDB support STABS production and consumption on a best effort basis only.

Within these ELF files, the GCC debuginfo level is also used. The default is level 2, where macro information is not present; level 3 has C/C++ macro definitions included, but the debuginfo can be very large with this setting. The command for the default `gcc -g` is the same as `gcc -g2`. To change the macro information to level three, use `gcc -g3`.

There are multiple levels of debuginfo available. Use the command `readelf -WS file` to see which sections are used in a file.

Table 5.1. debuginfo levels

Binary State	Command	Notes
Stripped	<code>strip file</code> or <code>gcc -s -o file</code>	Only the symbols required for runtime linkage with shared libraries are present. ELF section in use: <code>.dynsym</code>
ELF symbols	<code>gcc -o file</code>	Only the names of functions and variables are present, no binding to the source files and no types. ELF section in use: <code>.symtab</code>

Binary State	Command	Notes
DWARF debuginfo with macros	gcc -g -o file	The source file names and line numbers are known, including types. ELF section in use: .debug_*
DWARF debuginfo with macros	gcc -g3 -o file	Similar to gcc -g but the macros are known to GDB. ELF section in use: .debug_macro



Note

GDB never interprets the source files, it only displays them as text. Use **gcc -g** and its variants to store the information into DWARF.

Compiling a program or library with **gcc -rdynamic** is discouraged. For specific symbols, use **gcc -Wl,--dynamic-list=...** instead. If **gcc -rdynamic** is used, the **strip** command or **-s** gcc option have no effect. This is because all ELF symbols are kept in the binary for possible runtime linkage with shared libraries.

ELF symbols can be read by the **readelf -s file** command.

DWARF symbols are read by the **readelf -w file** command.

The command **readelf -wi file** is a good verification of debuginfo, compiled within your program. The commands **strip file** or **gcc -s** are commonly accidentally executed on the output during various compilation stages of the program.

The **readelf -w file** command can also be used to show a special section called **.eh_frame** with a format and purpose is similar to the DWARF section **.debug_frame**. The **.eh_frame** section is used for runtime C++ exception resolution and is present even if **-g** gcc option was not used. It is kept in the primary RPM and is never present in the debuginfo RPMs.

Debuginfo RPMs contain the sections **.symtab** and **.debug_***. Neither **.eh_frame**, **.eh_frame_hdr**, nor **.dynsym** are moved or present in debuginfo RPMs as those sections are needed during program runtime.

5.2. Installing Debuginfo Packages

Red Hat Enterprise Linux also provides **-debuginfo** packages for all architecture-dependent RPMs included in the operating system. A **packagename-debuginfo-version-release.architecture.rpm** package contains detailed information about the relationship of the package source files and the final installed binary. The debuginfo packages contain both **.debug** files, which in turn contain DWARF debuginfo and the source files used for compiling the binary packages.

**Note**

Most of the debugger functionality is missed if attempting to debug a package without having its debuginfo equivalent installed. For example, the names of exported shared library functions will still be available, but the matching source file lines will not be without the debuginfo package installed.

Use **gcc** compilation option **-g** for your own programs. The debugging experience is better if no optimizations (gcc option **-O**, such as **-O2**) is applied with **-g**.

For Red Hat Enterprise Linux 6, the debuginfo packages are now available on a new channel on the Red Hat Network. To install the **-debuginfo** package of a package (that is, typically **packagename-debuginfo**), first the machine has to be subscribed to the corresponding Debuginfo channel. For example, for Red Hat Enterprise Server 6, the corresponding channel would be **Red Hat Enterprise Linux Server Debuginfo (v. 6)**.

Red Hat Enterprise Linux system packages are compiled with optimizations (gcc option **-O2**). This means that some variables will be displayed as **<optimized out>**. Stepping through code will 'jump' a little but a crash can still be analyzed. If some debugging information is missing because of the optimizations, the right variable information can be found by disassembling the code and matching it to the source manually. This is applicable only in exceptional cases and is not suitable for regular debugging.

For system packages, GDB informs the user if it is missing some debuginfo packages that limit its functionality.

```
gdb ls
[...]
Reading symbols from /usr/bin/ls...(no debugging symbols found)...done.
Missing separate debuginfos, use: debuginfo-install coreutils-8.4-16.el6.x86_64
(gdb) q
```

If the system package to be debugged is known, use the command suggested by GDB above. It will also automatically install all the debug packages *packagename* depends on.

```
# debuginfo-install packagename
```

5.2.1. Installing Debuginfo Packages for Core Files Analysis

A core file is a representation of the memory image at the time of a process crash. For bug reporting of system program crashes, Red Hat recommends the use of the ABRT tool, explained in the *Automatic Bug Reporting Tool* chapter in the *Red Hat Deployment Guide*. If ABRT is not suitable for your purposes, the steps it automates are explained here.

If the **ulimit -c unlimited** setting is in use when a process crashes, the core file is dumped into the current directory. The core file contains only the memory areas modified by the process from the original state of disk files. In order to perform a full analysis of a crash, a core file is required to have:

- ▶ the core file itself
- ▶ the executable binary which has crashed, such as **/usr/sbin/sendmail**
- ▶ all the shared libraries loaded in the binary when it crashed
- ▶ .debug files and source files (both stored in debuginfo RPMs) for the executable and all of its loaded libraries

For a proper analysis, either the exact **version-release.architecture** for all the RPMs involved or the same build of your own compiled binaries is needed. At the time of the crash, the application may have already recompiled or been updated by **yum** on the disk, rendering the files inappropriate for the core file analysis.

The core file contains build-ids of all the binaries involved. For more information on build-id, see [Section 4.6, “build-id Unique Identification of Binaries”](#). The contents of the core file can be displayed by:

```
$ eu-unstrip -n --core=./core.9814
0x400000+0x207000 2818b2009547f780a5639c904cded443e564973e@0x400284 usr/bin/sleep
/usr/lib/debug/bin/sleep.debug [exe]
0x7fff26fff000+0x1000 1e2a683b7d877576970e4275d41a6aaec280795e@0x7fff26fff340 . -
linux-vdso.so.1
0x35e7e00000+0x3b6000 374add1ead31ccb449779bc7ee7877de3377e5ad@0x35e7e00280
/usr/lib64/libc-2.14.90.so /usr/lib/debug/lib64/libc-2.14.90.so.debug libc.so.6
0x35e7a00000+0x224000 3ed9e61c2b7e707ce244816335776afa2ad0307d@0x35e7a001d8
/usr/lib64/ld-2.14.90.so /usr/lib/debug/lib64/ld-2.14.90.so.debug ld-linux-x86-
64.so.2
```

The meaning of the columns in each line are:

- ▶ The in-memory address where the specific binary was mapped to (for example, **0x400000** in the first line).
- ▶ The size of the binary (for example, **+0x207000** in the first line).
- ▶ The 160-bit SHA-1 build-id of the binary (for example, **2818b2009547f780a5639c904cded443e564973e** in the first line).
- ▶ The in-memory address where the build-id bytes were stored (for example, **@0x400284** in the first line).
- ▶ The on-disk binary file, if available (for example, **usr/bin/sleep** in the first line). This was found by **eu-unstrip** for this module.
- ▶ The on-disk debuginfo file, if available (for example, **/usr/lib/debug/bin/sleep.debug**). However, best practice is to use the binary file reference instead.
- ▶ The shared library name as stored in the shared library list in the core file (for example, **libc.so.6** in the third line).

For each build-id (for example, **ab/cdef0123456789012345678901234567890123**) a symbolic link is included in its debuginfo RPM. Using the **/usr/bin/sleep** executable above as an example, the **coreutils-debuginfo** RPM contains, among other files:

```
lrwxrwxrwx 1 root root 24 Nov 29 17:07 /usr/lib/debug/.build-
id/28/18b2009547f780a5639c904cded443e564973e -> ../../../../../../bin/sleep*
lrwxrwxrwx 1 root root 21 Nov 29 17:07 /usr/lib/debug/.build-
id/28/18b2009547f780a5639c904cded443e564973e.debug -> ../../bin/sleep.debug
```

In some cases (such as loading a core file), GDB does not know the name, version, or release of a **name-debuginfo-version-release.rpm** package; it only knows the build-id. In such cases, GDB suggests a different command:

```
gdb -c ./core
[...]
Missing separate debuginfo for the main executable filename
Try: yum --disablerepo='*' --enablerepo='*debug*' install /usr/lib/debug/.build-id/ef/dd0b5e69b0742fa5e5bad0771df4d1df2459d1
```

The *version-release.architecture* of the binary package *packagename-debuginfo-version-release.architecture.rpm* must be an exact match. If it differs then GDB cannot use the debuginfo package. Even the same *version-release.architecture* from a different build leads to an incompatible debuginfo package. If GDB reports a missing debuginfo, ensure to recheck:

```
rpm -q packagename packagename-debuginfo
```

The *version-release.architecture* definitions should match.

```
rpm -V packagename packagename-debuginfo
```

This command should produce no output, except possibly modified configuration files of *packagename*, for example.

```
rpm -qi packagename packagename-debuginfo
```

The *version-release.architecture* should display matching information for Vendor, Build Date, and Build Host. For example, using a CentOS debuginfo RPM for a Red Hat Enterprise Linux RPM package will not work.

If the required build-id is known, the following command can query which RPM contains it:

```
$ repoquery --disablerepo='*' --enablerepo='*-debug*' -qf /usr/lib/debug/.build-id/ef/dd0b5e69b0742fa5e5bad0771df4d1df2459d1
```

For example, a version of an executable which matches the core file can be installed by:

```
# yum --enablerepo='*-debug*' install $(eu-unstrip -n --core=./core.9814 | sed -e 's#^[^ ]* \((..\)\([^\@ ]*\)\.*$#/usr/lib/debug/.build-id/\1/\2#p' -e 's/$/.debug/')
```

Similar methods are available if the binaries are not packaged into RPMs and stored in yum repositories. It is possible to create local repositories with custom application builds by using **/usr/bin/createrepo**.

5.3. GDB

Fundamentally, like most debuggers, GDB manages the execution of compiled code in a very closely controlled environment. This environment makes possible the following fundamental mechanisms necessary to the operation of GDB:

- ▶ Inspect and modify memory within the code being debugged (for example, reading and setting variables).
- ▶ Control the execution state of the code being debugged, principally whether it's running or stopped.
- ▶ Detect the execution of particular sections of code (for example, stop running code when it reaches a specified area of interest to the programmer).
- ▶ Detect access to particular areas of memory (for example, stop running code when it accesses a specified variable).

- ▶ Execute portions of code (from an otherwise stopped program) in a controlled manner.
- ▶ Detect various programmatic asynchronous events such as signals.

The operation of these mechanisms rely mostly on information produced by a compiler. For example, to view the value of a variable, GDB has to know:

- ▶ The location of the variable in memory
- ▶ The nature of the variable

This means that displaying a double-precision floating point value requires a very different process from displaying a string of characters. For something complex like a structure, GDB has to know not only the characteristics of each individual elements in the structure, but the morphology of the structure as well.

GDB requires the following items in order to fully function:

Debug Information

Much of GDB's operations rely on a program's *debug information*. While this information generally comes from compilers, much of it is necessary only while debugging a program, that is, it is not used during the program's normal execution. For this reason, compilers do not always make that information available by default — GCC, for instance, must be explicitly instructed to provide this debugging information with the `-g` flag.

To make full use of GDB's capabilities, it is *highly advisable* to make the debug information available first to GDB. GDB can only be of *very limited* use when run against code with no available debug information.

Source Code

One of the most useful features of GDB (or any other debugger) is the ability to associate events and circumstances in program execution with their corresponding location in source code. This location normally refers to a specific line or series of lines in a source file. This, of course, would require that a program's source code be available to GDB at debug time.

5.3.1. Simple GDB

GDB literally contains dozens of commands. This section describes the most fundamental ones.

br (breakpoint)

The breakpoint command instructs GDB to halt execution upon reaching a specified point in the execution. That point can be specified a number of ways, but the most common are just as the line number in the source file, or the name of a function. Any number of breakpoints can be in effect simultaneously. This is frequently the first command issued after starting GDB.

r (run)

The `run` command starts the execution of the program. If `run` is executed with any arguments, those arguments are passed on to the executable as if the program has been started normally. Users normally issue this command after setting breakpoints.

Before an executable is started, or once the executable stops at, for example, a breakpoint, the state of many aspects of the program can be inspected. The following commands are a few of the more common ways things can be examined.

p (print)

The **print** command displays the value of the argument given, and that argument can be almost anything relevant to the program. Usually, the argument is the name of a variable of any complexity, from a simple single value to a structure. An argument can also be an expression valid in the current language, including the use of program variables and library functions, or functions defined in the program being tested.

bt (backtrace)

The **backtrace** displays the chain of function calls used up until the execution was terminated. This is useful for investigating serious bugs (such as segmentation faults) with elusive causes.

l (list)

When execution is stopped, the **list** command shows the line in the source code corresponding to where the program stopped.

The execution of a stopped program can be resumed in a number of ways. The following are the most common.

c (continue)

The **continue** command restarts the execution of the program, which will continue to execute until it encounters a breakpoint, runs into a specified or emergent condition (for example, an error), or terminates.

n (next)

Like **continue**, the **next** command also restarts execution; however, in addition to the stopping conditions implicit in the **continue** command, **next** will also halt execution at the next sequential line of code in the current source file.

s (step)

Like **next**, the **step** command also halts execution at each sequential line of code in the current source file. However, if execution is currently stopped at a source line containing a *function call*, GDB stops execution after entering the function call (rather than executing it).

fini (finish)

Like the aforementioned commands, the **finish** command resumes executions, but halts when execution returns from a function.

Finally, two essential commands:

q (quit)

This terminates the execution.

h (help)

The **help** command provides access to its extensive internal documentation. The command takes arguments: **help breakpoint** (or **h br**), for example, shows a detailed description of the **breakpoint** command. Refer to the **help** output of each command for more detailed information.

5.3.2. Running GDB

This section will describe a basic execution of GDB, using the following simple program:

hello.c

```
#include <stdio.h>

char hello[] = { "Hello, World!" };

int
main()
{
    fprintf (stdout, "%s\n", hello);
    return (0);
}
```

The following procedure illustrates the debugging process in its most basic form.

Procedure 5.1. Debugging a 'Hello World' Program

1. Compile [hello.c](#) into an executable with the debug flag set, as in:

```
gcc -g -o hello hello.c
```

Ensure that the resulting binary **hello** is in the same directory as **hello.c**.

2. Run **gdb** on the **hello** binary, that is, **gdb hello**.
3. After several introductory comments, **gdb** will display the default GDB prompt:

```
(gdb)
```

4. The variable **hello** is global, so it can be seen even before the **main** procedure starts:

```
(gdb) p hello
$1 = "Hello, World!"
(gdb) p hello[0]
$2 = 72 'H'
(gdb) p *hello
$3 = 72 'H'
(gdb)
```

Note that the **print** targets **hello[0]** and ***hello** require the evaluation of an expression, as does, for example, ***(hello + 1)**:

```
(gdb) p *(hello + 1)
$4 = 101 'e'
```

5. Next, list the source:

```
(gdb) l
1      #include <stdio.h>
2
3      char hello[] = { "Hello, World!" };
4
5      int
```

```

6     main()
7     {
8         fprintf (stdout, "%s\n", hello);
9         return (0);
10    }

```

The **list** reveals that the **fprintf** call is on line 8. Apply a breakpoint on that line and resume the code:

```

(gdb) br 8
Breakpoint 1 at 0x80483ed: file hello.c, line 8.
(gdb) r
Starting program: /home/moller/tinkering/gdb-manual/hello

Breakpoint 1, main () at hello.c:8
8         fprintf (stdout, "%s\n", hello);

```

6. Finally, use the **next** command to step past the **fprintf** call, executing it:

```

(gdb) n
Hello, World!
9         return (0);

```

The following sections describe more complex applications of GDB.

5.3.3. Conditional Breakpoints

In many real-world cases, a program may perform its task well during the first few thousand times; it may then start crashing or encountering errors during its eight thousandth iteration of the task. Debugging programs like this can be difficult, as it is hard to imagine a programmer with the patience to issue a **continue** command thousands of times just to get to the iteration that crashed.

Situations like this are common in real life, which is why GDB allows programmers to attach conditions to a breakpoint. For example, consider the following program:

simple.c

```

#include <stdio.h>

main()
{
    int i;

    for (i = 0;; i++) {
        fprintf (stdout, "i = %d\n", i);
    }
}

```

To set a conditional breakpoint at the GDB prompt:

```

(gdb) br 8 if i == 8936
Breakpoint 1 at 0x80483f5: file iterations.c, line 8.
(gdb) r

```

With this condition, the program execution will eventually stop with the following output:

```

i = 8931
i = 8932
i = 8933
i = 8934
i = 8935

Breakpoint 1, main () at iterations.c:8
8      fprintf (stdout, "i = %d\n", i);

```

Inspect the breakpoint information (using **info br**) to review the breakpoint status:

```

(gdb) info br
Num      Type          Disp Enb Address      What
1        breakpoint    keep y   0x080483f5 in main at iterations.c:8
          stop only if i == 8936
          breakpoint already hit 1 time

```

5.3.4. Forked Execution

Among the more challenging bugs confronting programmers is where one program (the *parent*) makes an independent copy of itself (a *fork*). That fork then creates a *child* process which, in turn, fails. Debugging the parent process may or may not be useful. Often the only way to get to the bug may be by debugging the child process, but this is not always possible.

The **set follow-fork-mode** feature is used to overcome this barrier allowing programmers to follow a child process instead of the parent process.

set follow-fork-mode parent

The original process is debugged after a fork. The child process runs unimpeded. This is the default.

set follow-fork-mode child

The new process is debugged after a fork. The parent process runs unimpeded.

show follow-fork-mode

Display the current debugger response to a fork call.

Use the **set detach-on-fork** command to debug both the parent and the child processes after a fork, or retain debugger control over them both.

set detach-on-fork on

The child process (or parent process, depending on the value of **follow-fork-mode**) will be detached and allowed to run independently. This is the default.

set detach-on-fork off

Both processes will be held under the control of GDB. One process (child or parent, depending on the value of **follow-fork-mode**) is debugged as usual, while the other is suspended.

show detach-on-fork

Show whether **detach-on-fork** mode is on or off.

Consider the following program:

fork.c

```
#include <unistd.h>

int main()
{
    pid_t pid;
    const char *name;

    pid = fork();
    if (pid == 0)
    {
        name = "I am the child";
    }
    else
    {
        name = "I am the parent";
    }
    return 0;
}
```

This program, compiled with the command `gcc -g fork.c -o fork -lpthread` and examined under GDB will show:

```
gdb ./fork
[...]
(gdb) break main
Breakpoint 1 at 0x4005dc: file fork.c, line 8.
(gdb) run
[...]
Breakpoint 1, main () at fork.c:8
8   pid = fork();
(gdb) next
Detaching after fork from child process 3840.
9   if (pid == 0)
(gdb) next
15      name = "I am the parent";
(gdb) next
17   return 0;
(gdb) print name
$1 = 0x400717 "I am the parent"
```

GDB followed the parent process and allowed the child process (process 3840) to continue execution.

The following is the same test using `set follow-fork-mode child`.

```
(gdb) set follow-fork-mode child
(gdb) break main
Breakpoint 1 at 0x4005dc: file fork.c, line 8.
(gdb) run
[...]
Breakpoint 1, main () at fork.c:8
8   pid = fork();
(gdb) next
[New process 3875]
[Thread debugging using libthread_db enabled]
[Switching to Thread 0x7ffff7fd5720 (LWP 3875)]
9   if (pid == 0)
```

```
(gdb) next
11     name = "I am the child";
(gdb) next
17     return 0;
(gdb) print name
$2 = 0x400708 "I am the child"
(gdb)
```

GDB switched to the child process here.

This can be permanent by adding the setting to the appropriate `.gdbinit`.

For example, if `set follow-fork-mode ask` is added to `~/.gdbinit`, then ask mode becomes the default mode.

5.3.5. Debugging Individual Threads

GDB has the ability to debug individual threads, and to manipulate and examine them independently. This functionality is not enabled by default. To do so use `set non-stop on` and `set target-async on`. These can be added to `.gdbinit`. Once that functionality is turned on, GDB is ready to conduct thread debugging.

For example, the following program creates two threads. These two threads, along with the original thread executing main makes a total of three threads.

three-threads.c

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_t thread;

void* thread3 (void* d)
{
    int count3 = 0;

    while(count3 < 1000){
        sleep(10);
        printf("Thread 3: %d\n", count3++);
    }
    return NULL;
}

void* thread2 (void* d)
{
    int count2 = 0;

    while(count2 < 1000){
        printf("Thread 2: %d\n", count2++);
    }
    return NULL;
}

int main (){

    pthread_create (&thread, NULL, thread2, NULL);
    pthread_create (&thread, NULL, thread3, NULL);
```

```
//Thread 1
int count1 = 0;

while(count1 < 1000){
    printf("Thread 1: %d\n", count1++);
}

pthread_join(thread, NULL);
return 0;
}
```

Compile this program in order to examine it under GDB.

```
gcc -g three-threads.c -o three-threads -lpthread
gdb ./three-threads
```

First set breakpoints on all thread functions; thread1, thread2, and main.

```
(gdb) break thread3
Breakpoint 1 at 0x4006c0: file three-threads.c, line 9.
(gdb) break thread2
Breakpoint 2 at 0x40070c: file three-threads.c, line 20.
(gdb) break main
Breakpoint 3 at 0x40074a: file three-threads.c, line 30.
```

Then run the program.

```
(gdb) run
[...]
Breakpoint 3, main () at three-threads.c:30
30 pthread_create (&thread, NULL, thread2, NULL);
[...]
(gdb) info threads
* 1 Thread 0x7ffff7fd5720 (LWP 4620) main () at three-threads.c:30
(gdb)
```

Note that the command **info threads** provides a summary of the program's threads and some details about their current state. In this case there is only one thread that has been created so far.

Continue execution some more.

```
(gdb) next
[New Thread 0x7ffff7fd3710 (LWP 4687)]
31 pthread_create (&thread, NULL, thread3, NULL);
(gdb)
Breakpoint 2, thread2 (d=0x0) at three-threads.c:20
20 int count2 = 0;
next
[New Thread 0x7ffff75d2710 (LWP 4688)]
34 int count1 = 0;
(gdb)
Breakpoint 1, thread3 (d=0x0) at three-threads.c:9
9 int count3 = 0;
info threads
 3 Thread 0x7ffff75d2710 (LWP 4688) thread3 (d=0x0) at three-threads.c:9
 2 Thread 0x7ffff7fd3710 (LWP 4687) thread2 (d=0x0) at three-threads.c:20
* 1 Thread 0x7ffff7fd5720 (LWP 4620) main () at three-threads.c:34
```

Here, two more threads are created. The star indicates the thread currently under focus. Also, the newly created threads have hit the breakpoint set for them in their initialization functions. Namely, `thread2()` and `thread3()`.

To begin real thread debugging, use the `thread <thread number>` command to switch the focus to another thread.

```
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff7fd3710 (LWP 4687))]#0  thread2 (d=0x0)
    at three-threads.c:20
20  int count2 = 0;
(gdb) list
15  return NULL;
16  }
17
18 void* thread2 (void* d)
19 {
20  int count2 = 0;
21
22  while(count2 < 1000){
23  printf("Thread 2: %d\n", count2++);
24  }
```

Thread 2 stopped at line 20 in its function `thread2()`.

```
(gdb) next
22  while(count2 < 1000){
(gdb) print count2
$1 = 0
(gdb) next
23  printf("Thread 2: %d\n", count2++);
(gdb) next
Thread 2: 0
22  while(count2 < 1000){
(gdb) next
23  printf("Thread 2: %d\n", count2++);
(gdb) print count2
$2 = 1
(gdb) info threads
  3 Thread 0x7ffff75d2710 (LWP 4688)  thread3 (d=0x0) at three-threads.c:9
*  2 Thread 0x7ffff7fd3710 (LWP 4687)  thread2 (d=0x0) at three-threads.c:23
  1 Thread 0x7ffff7fd5720 (LWP 4620)  main () at three-threads.c:34
(gdb)
```

Above, a few lines of `thread2` printed the counter `count2` and left thread 2 at line 23 as is seen by the output of 'info threads'.

Now thread3.

```
(gdb) thread 3
[Switching to thread 3 (Thread 0x7ffff75d2710 (LWP 4688))]#0  thread3 (d=0x0)
    at three-threads.c:9
 9  int count3 = 0;
(gdb) list
 4
 5 pthread_t thread;
 6
 7 void* thread3 (void* d)
```

```

8 {
9   int count3 = 0;
10
11  while(count3 < 1000){
12    sleep(10);
13    printf("Thread 3: %d\n", count3++);
(gdb)

```

Thread three is a little different in that it has a sleep statement and executes slowly. Think of it as a representation of an uninteresting IO thread. Because this thread is uninteresting, continue its execution uninterrupted, using the **continue**.

```

(gdb) continue &
(gdb) Thread 3: 0
Thread 3: 1
Thread 3: 2
Thread 3: 3

```

Take note of the & at the end of the **continue**. This allows the GDB prompt to return so other commands can be executed. Using the **interrupt**, execution can be stopped should thread 3 become interesting again.

```

(gdb) interrupt
[Thread 0x7ffff75d2710 (LWP 4688)] #3 stopped.
0x000000343f4a6a6d in nanosleep () at ../sysdeps/unix/syscall-template.S:82
82 T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)

```

It is also possible to go back to the original main thread and examine it some more.

```

(gdb) thread 1
[Switching to thread 1 (Thread 0x7ffff7fd5720 (LWP 4620))]#0 main ()
    at three-threads.c:34
34  int count1 = 0;
(gdb) next
36  while(count1 < 1000){
(gdb) next
37    printf("Thread 1: %d\n", count1++);
(gdb) next
Thread 1: 0
36  while(count1 < 1000){
(gdb) next
37    printf("Thread 1: %d\n", count1++);
(gdb) next
Thread 1: 1
36  while(count1 < 1000){
(gdb) next
37    printf("Thread 1: %d\n", count1++);
(gdb) next
Thread 1: 2
36  while(count1 < 1000){
(gdb) print count1
$3 = 3
(gdb) info threads
   3 Thread 0x7ffff75d2710 (LWP 4688) 0x000000343f4a6a6d in nanosleep ()
     at ../sysdeps/unix/syscall-template.S:82
   2 Thread 0x7ffff7fd3710 (LWP 4687) thread2 (d=0x0) at three-threads.c:23
*  1 Thread 0x7ffff7fd5720 (LWP 4620) main () at three-threads.c:36
(gdb)

```

As can be seen from the output of info threads, the other threads are where they were left, unaffected by the debugging of thread 1.

5.3.6. Alternative User Interfaces for GDB

GDB uses the command line as its default interface. However, it also has an API called *machine interface* (MI). MI allows IDE developers to create other user interfaces to GDB.

Some examples of these interfaces are:

Eclipse (CDT)

A graphical debugger interface integrated with the Eclipse development environment. More information can be found at the [Eclipse website](#).

Nemiver

A graphical debugger interface which is well suited to the GNOME Desktop Environment. More information can be found at the [Nemiver website](#)

Emacs

A GDB interface which is integrated with the emacs. More information can be found at the [Emacs website](#)

5.3.7. GDB Documentation

For more detailed information about GDB, refer to the GDB manual:

<http://sources.redhat.com/gdb/current/onlinedocs/gdb.html>

Also, the commands `info gdb` and `man gdb` will provide more concise information that is up to date with the installed version of gdb.

5.4. Variable Tracking at Assignments

Variable Tracking at Assignments (VTA) is a new infrastructure included in GCC used to improve variable tracking during optimizations. This allows GCC to produce more precise, meaningful, and useful debugging information for GDB, SystemTap, and other debugging tools.

When GCC compiles code with optimizations enabled, variables are renamed, moved around, or even removed altogether. As such, optimized compiling can cause a debugger to report that some variables have been <optimized out>. With VTA enabled, optimized code is internally annotated to ensure that optimization passes to transparently keep track of each variable's value, regardless of whether the variable is moved or removed. The effect of this is more parameter and variable values available, even for the optimized (`gcc -O2 -g` built) code. It also displays the <optimized out> message less.

VTA's benefits are more pronounced when debugging applications with inlined functions. Without VTA, optimization could completely remove some arguments of an inlined function, preventing the debugger from inspecting its value. With VTA, optimization will still happen, and appropriate debugging information will be generated for any missing arguments.

VTA is enabled by default when compiling code with optimizations and debugging information enabled (that is, `gcc -O -g` or, more commonly, `gcc -O2 -g`). To disable VTA during such builds, add the `-fno-var-tracking-assignments`. In addition, the VTA infrastructure includes the new `gcc` option `-fcompare-debug`. This option tests code compiled by GCC with debug information and without debug

information: the test passes if the two binaries are identical. This test ensures that executable code is not affected by any debugging options, which further ensures that there are no hidden bugs in the debug code. Note that **-fcompare-debug** adds significant cost in compilation time. Refer to **man gcc** for details about this option.

For more information about the infrastructure and development of VTA, refer to *A Plan to Fix Local Variable Debug Information in GCC*, available at the following link:

http://gcc.gnu.org/wiki/Var_Tracking_Assignments

A slide deck version of this whitepaper is also available at <http://people.redhat.com/aoliva/papers/vta/slides.pdf>.

5.5. Python Pretty-Printers

The GDB command **print** outputs comprehensive debugging information for a target application. GDB aims to provide as much debugging data as it can to users; however, this means that for highly complex programs the amount of data can become very cryptic.

In addition, GDB does not provide any tools that help decipher GDB **print** output. GDB does not even empower users to easily create tools that can help decipher program data. This makes the practice of reading and understanding debugging data quite arcane, particularly for large, complex projects.

For most developers, the only way to customize GDB **print** output (and make it more meaningful) is to revise and recompile GDB. However, very few developers can actually do this. Further, this practice will not scale well, particularly if the developer must also debug other programs that are heterogeneous and contain equally complex debugging data.

To address this, the Red Hat Enterprise Linux version of GDB is now compatible with Python *pretty-printers*. This allows the retrieval of more meaningful debugging data by leaving the introspection, printing, and formatting logic to a *third-party* Python script.

Compatibility with Python pretty-printers gives you the chance to truly customize GDB output as you see fit. This makes GDB a more viable debugging solution to a wider range of projects, since you now have the flexibility to *adapt* GDB output as required, and with greater ease. Further, developers with intimate knowledge of a project and a specific programming language are best qualified in deciding what kind of output is meaningful, allowing them to improve the usefulness of that output.

The Python pretty-printers implementation allows users to automatically inspect, format, and print program data according to specification. These specifications are written as rules implemented via Python scripts. This offers the following benefits:

Safe

To pass program data to a set of registered Python pretty-printers, the GDB development team added *hooks* to the GDB printing code. These hooks were implemented with safety in mind: the built-in GDB printing code is still intact, allowing it to serve as a default fallback printing logic. As such, if no specialized printers are available, GDB will still print debugging data the way it always did. This ensures that GDB is backwards-compatible; users who do not require pretty-printers can still continue using GDB.

Highly Customizable

This new "Python-scripted" approach allows users to distill as much knowledge as required into specific printers. As such, a project can have an entire library of printer scripts that parses program data in a unique manner specific to its user's requirements. There is no limit to the number of printers a user can build for a specific project; what's more, being able to customize debugging data script by script offers

users an easier way to re-use and re-purpose printer scripts — or even a whole library of them.

Easy to Learn

The best part about this approach is its lower barrier to entry. Python scripting is comparatively easy to learn and has a large library of free documentation available online. In addition, most programmers already have basic to intermediate experience in Python scripting, or in scripting in general.

Here is a small example of a pretty printer. Consider the following C++ program:

fruit.cc

```
enum Fruits {Orange, Apple, Banana};

class Fruit
{
    int fruit;

public:
    Fruit (int f)
    {
        fruit = f;
    }
};

int main()
{
    Fruit myFruit(Apple);
    return 0;           // line 17
}
```

This is compiled with the command `g++ -g fruit.cc -o fruit`. Now, examine this program with GDB.

```
gdb ./fruit
[...]
(gdb) break 17
Breakpoint 1 at 0x40056d: file fruit.cc, line 17.
(gdb) run

Breakpoint 1, main () at fruit.cc:17
17  return 0;           // line 17
(gdb) print myFruit
$1 = {fruit = 1}
```

The output of `{fruit = 1}` is correct because that is the internal representation of 'fruit' in the data structure 'Fruit'. However, this is not easily read by humans as it is difficult to tell which fruit the integer 1 represents.

To solve this problem, write the following pretty printer:

```
fruit.py

class FruitPrinter:
    def __init__(self, val):
        self.val = val

    def to_string (self):
        fruit = self.val['fruit']
```

```

    if (fruit == 0):
        name = "Orange"
    elif (fruit == 1):
        name = "Apple"
    elif (fruit == 2):
        name = "Banana"
    else:
        name = "unknown"
    return "Our fruit is " + name

def lookup_type (val):
    if str(val.type) == 'Fruit':
        return FruitPrinter(val)
    return None

gdb.pretty_printers.append (lookup_type)

```

Examine this printer from the bottom up.

The line `gdb.pretty_printers.append (lookup_type)` adds the function `lookup_type` to GDB's list of printer lookup functions.

The function `lookup_type` is responsible for examining the type of object to be printed, and returning an appropriate pretty printer. The object is passed by GDB in the parameter `val`. `val.type` is an attribute which represents the type of the pretty printer.

`FruitPrinter` is where the actual work is done. More specifically in the `to_string` function of that Class. In this function, the integer `fruit` is retrieved using the python dictionary syntax `self.val['fruit']`. Then the name is determined using that value. The string returned by this function is the string that will be printed to the user.

After creating `fruit.py`, it must then be loaded into GDB with the following command:

```
(gdb) python execfile("fruit.py")
```

The *GDB and Python Pretty-Printers* whitepaper provides more details on this feature. This whitepaper also includes details and examples on how to write your own Python pretty-printer as well as how to import it into GDB. Refer to the following link for more information:

<http://sourceware.org/gdb/onlinedocs/gdb/Pretty-Printing.html>

5.6. Debugging C/C++ Applications with Eclipse

The Eclipse C/C++ development tools have excellent integration with the GNU Debugger (GDB). These Eclipse plug-ins take advantage of the latest features available in GDB.

Starting a debugging session for an application is similar to launching the application through either the context menu's **Debug As** → **C/C++ Application**, or using the **Run** menu. The context menu can be accessed in one of three ways:

- ▶ Clicking the right mouse button with the cursor in the editor.
- ▶ On the application binary.
- ▶ On the project containing the binary of interest.

If more than one binary can be launched, a dialog will be presented to choose which one.

After the session has started, a prompt will appear to switch to the Debug perspective, which contains the following collection of views related to debugging.

Control View

The Control View is known as the Debug view and has buttons for stepping over and into code selections. It also allows for thread process suspension.

Source Code Editor View

The Source Code Editor View reflects which source code lines correspond to the position of the debugger in the execution. By pressing the **Instruction Stepping Mode** button in the Debug view toolbar, it is possible to control the execution of the application by assembly instruction instead of by source code line.

Console View

The Console View displays the input and output that is available.

Finally, variable data and other information can be found in the corresponding views in the Debug perspective.

For further details, refer to the **Concepts → Debug, Getting Started → Debugging Projects**, and **Tasks → Running and Debugging Projects** sections of the *C/C++ Development User Guide* in the Help Contents.

Chapter 6. Profiling

Developers profile programs to focus attention on the areas of the program that have the largest impact on performance. The types of data collected include what section of the program consumes the most processor time, and where memory is allocated. Profiling collects data from the actual program execution. Thus, the quality of the data collect is influenced by the actual tasks being performed by the program. The tasks performed during profiling should be representative of actual use; this ensures that problems arising from realistic use of the program are addressed during development.

Red Hat Enterprise Linux includes a number of different tools (**Valgrind**, **OProfile**, **perf**, and **SystemTap**) to collect profiling data. Each tool is suitable for performing specific types of profile runs, as described in the following sections.

6.1. Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools that can be used to profile applications in detail. **Valgrind** tools are generally used to automatically detect many memory management and threading problems. The **Valgrind** suite also includes tools that allow the building of new profiling tools as required.

Valgrind provides instrumentation for user-space binaries to check for errors, such as the use of uninitialized memory, improper allocation/freeing of memory, and improper arguments for systemcalls. Its profiling tools can be used by normal users on most binaries; however, compared to other profilers, **Valgrind** profile runs are significantly slower. To profile a binary, **Valgrind** rewrites its executable and instruments the rewritten binary. **Valgrind**'s tools are most useful for looking for memory-related issues in user-space programs; it is not suitable for debugging time-specific issues or kernel-space instrumentation/debugging.

Previously, **Valgrind** did not support IBM System z architecture. However, as of 6.1, this support has been added, meaning **Valgrind** now supports all hardware architectures that are supported by Red Hat Enterprise Linux 6.x.

6.1.1. Valgrind Tools

The **Valgrind** suite is composed of the following tools:

memcheck

This tool detects memory management problems in programs by checking all reads from and writes to memory and intercepting all system calls to **malloc**, **new**, **free**, and **delete**.

memcheck is perhaps the most used **Valgrind** tool, as memory management problems can be difficult to detect using other means. Such problems often remain undetected for long periods, eventually causing crashes that are difficult to diagnose.

cachegrind

cachegrind is a cache profiler that accurately pinpoints sources of cache misses in code by performing a detailed simulation of the L1, D1 and L2 caches in the CPU. It shows the number of cache misses, memory references, and instructions accruing to each line of source code; **cachegrind** also provides per-function, per-module, and whole-program summaries, and can even show counts for each individual machine instructions.

callgrind

Like **cachegrind**, **callgrind** can model cache behavior. However, the main purpose of **callgrind** is to record callgraphs data for the executed code.

Example: How to record call-graphic data for the executed code.

massif

massif is a heap profiler; it measures how much heap memory a program uses, providing information on heap blocks, heap administration overheads, and stack sizes. Heap profilers are useful in finding ways to reduce heap memory usage. On systems that use virtual memory, programs with optimized heap memory usage are less likely to run out of memory, and may be faster as they require less paging.

helgrind

In programs that use the POSIX pthreads threading primitives, **helgrind** detects synchronization errors. Such errors are:

- ▶ Misuses of the POSIX pthreads API
- ▶ Potential deadlocks arising from lock ordering problems
- ▶ Data races (that is, accessing memory without adequate locking)

Valgrind also allows you to develop your own profiling tools. In line with this, **Valgrind** includes the **lackey** tool, which is a sample that can be used as a template for generating your own tools.

6.1.2. Using Valgrind

The **valgrind** package and its dependencies install all the necessary tools for performing a **Valgrind** profile run. To profile a program with **Valgrind**, use:

```
valgrind --tool=toolname program
```

Refer to [Section 6.1.1, “Valgrind Tools”](#) for a list of arguments for **toolname**. In addition to the suite of **Valgrind** tools, **none** is also a valid argument for **toolname**; this argument allows you to run a program under **Valgrind** without performing any profiling. This is useful for debugging or benchmarking **Valgrind** itself.

You can also instruct **Valgrind** to send all of its information to a specific file. To do so, use the option **--log-file=filename**. For example, to check the memory usage of the executable file **hello** and send profile information to **output**, use:

```
valgrind --tool=memcheck --log-file=output hello
```

Refer to [Section 6.1.4, “Valgrind Documentation”](#) for more information on **Valgrind**, along with other available documentation on the **Valgrind** suite of tools.

6.1.3. Valgrind Plug-in for Eclipse

The **Valgrind** plug-in for Eclipse integrates several **Valgrind** tools into Eclipse. This allows Eclipse users to seamlessly include profiling capabilities into their workflow. At present, the **Valgrind** plug-in for Eclipse supports three **Valgrind** tools:

- ▶ **Memcheck**
- ▶ **Massif**
- ▶ **Cachegrind**

To launch a **Valgrind** profile run, navigate to **Run > Profile**. This will open the **Profile As** dialog, from which you can select a tool for a profile run.

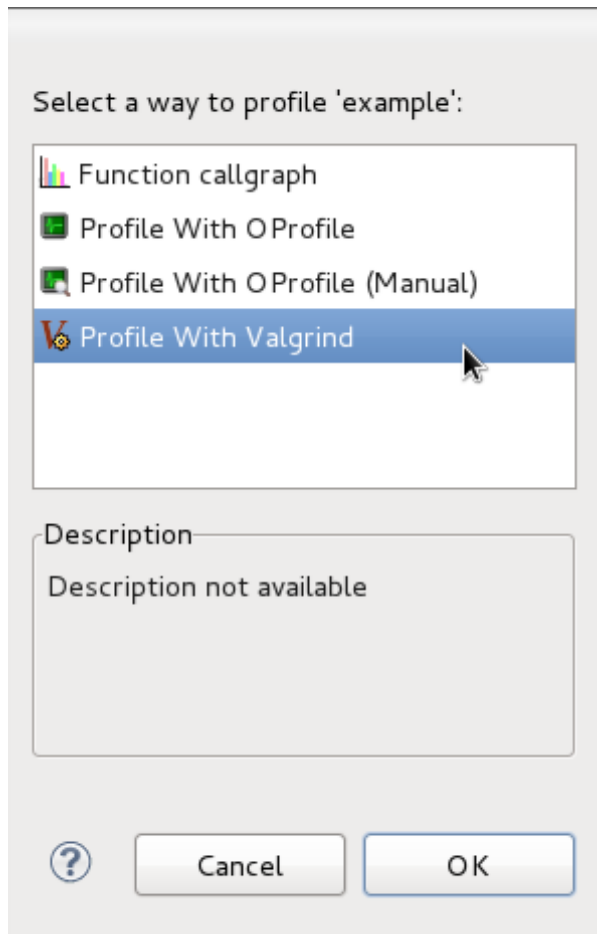


Figure 6.1. Profile As

To configure each tool for a profile run, navigate to **Run > Profile Configuration**. This will open the **Profile Configuration** menu.

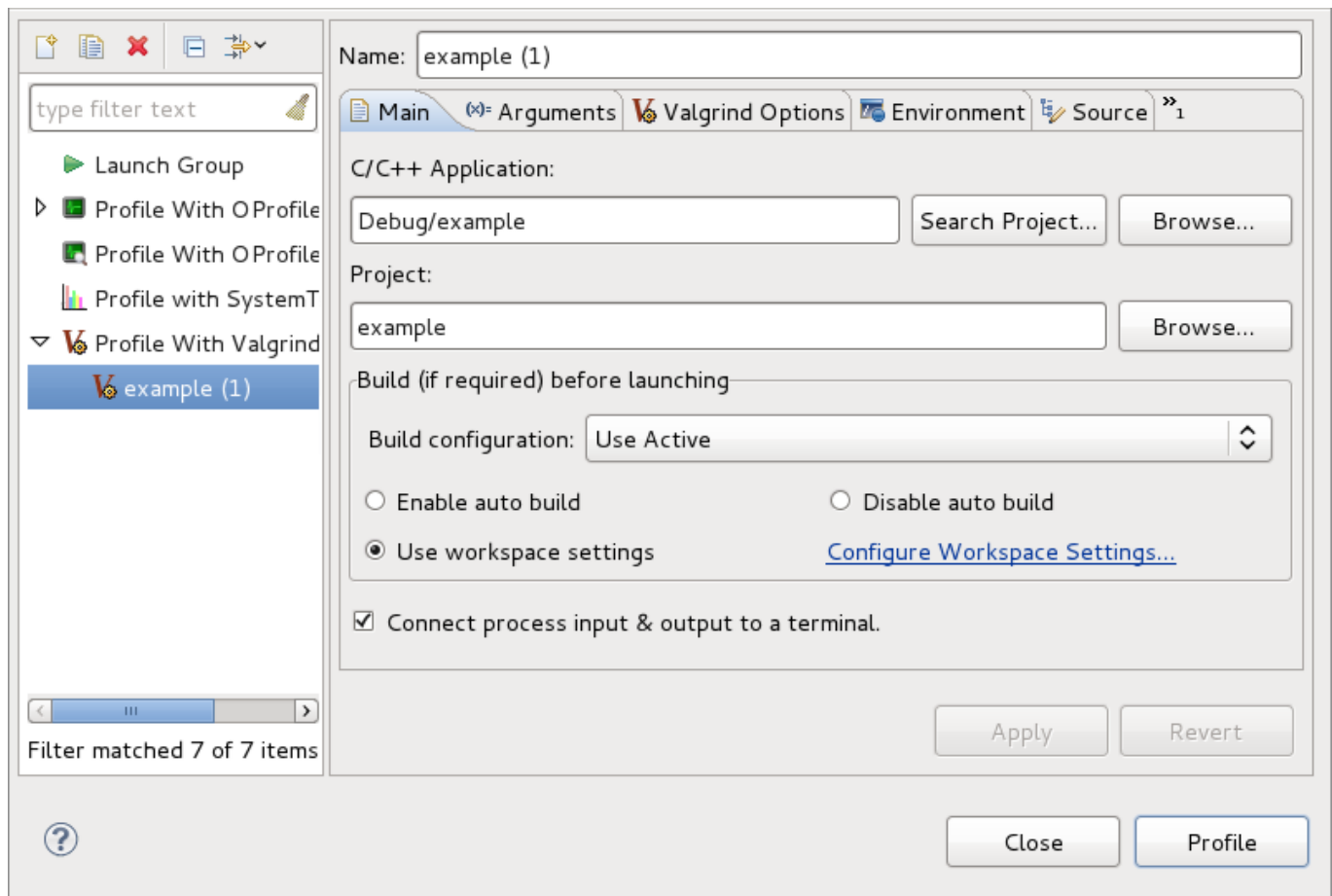


Figure 6.2. Profile Configuration

The **Valgrind** plug-in for Eclipse is provided by the **eclipse-valgrind** package. For more information about this plug-in, refer to *Valgrind Integration User Guide* in the Eclipse **Help Contents**.

6.1.4. Valgrind Documentation

For more extensive information on **Valgrind**, refer to **man valgrind**. Red Hat Enterprise Linux 6 also provides a comprehensive *Valgrind Documentation* book, available as PDF and HTML in:

- ▶ **file:///usr/share/doc/valgrind-version/valgrind_manual.pdf**
- ▶ **file:///usr/share/doc/valgrind-version/html/index.html**

The *Valgrind Integration User Guide* in the Eclipse **Help Contents** also provides detailed information on the setup and usage of the **Valgrind** plug-in for Eclipse. This guide is provided by the **eclipse-valgrind** package.

6.2. OProfile

OProfile is a system-wide Linux profiler, capable of running at low overhead. It consists of a kernel driver and a daemon for collecting raw sample data, along with a suite of tools for parsing that data into meaningful information. OProfile is generally used by developers to determine which sections of code consume the most amount of CPU time, and why.

During a profile run, OProfile uses the processor's performance monitoring hardware. **Valgrind** rewrites the binary of an application, and in turn instruments it. OProfile, on the other hand, profiles a running application as-is. It sets up the performance monitoring hardware to take a sample every *x* number of events (for example, cache misses or branch instructions). Each sample also contains information on where it occurred in the program.

OProfile's profiling methods consume less resources than **Valgrind**. However, OProfile requires root privileges. OProfile is useful for finding "hot-spots" in code, and looking for their causes (for example, poor cache performance, branch mispredictions).

Using OProfile involves starting the OProfile daemon (**oprofiled**), running the program to be profiled, collecting the system profile data, and parsing it into a more understandable format. OProfile provides several tools for every step of this process.

6.2.1. OProfile Tools

The most useful OProfile commands include the following:

perf

New in Red Hat Enterprise Linux 7, **perf** uses the Linux Performance Events subsystem, and so can completely replace use of the **opcontrol** daemon. See the *Red Hat Enterprise Linux 7 System Administrator's Guide* for further details.

opcontrol

This tool is used to start/stop the OProfile daemon and configure a profile session.

opreport

The **opreport** command outputs binary image summaries, or per-symbol data, from OProfile profiling sessions.

opannotate

The **opannotate** command outputs annotated source and/or assembly from the profile data of an OProfile session.

oparchive

The **oparchive** command generates a directory populated with executable, debug, and OProfile sample files. This directory can be moved to another machine (via **tar**), where it can be analyzed offline.

opgprof

Like **opreport**, the **opgprof** command outputs profile data for a given binary image from an OProfile session. The output of **opgprof** is in **gprof** format.

For a complete list of OProfile commands, refer to **man oprofile**. For detailed information on each OProfile command, refer to its corresponding **man** page. Refer to [Section 6.2.4, "OProfile Documentation"](#) for other available documentation on OProfile.

6.2.2. Using OProfile

The **oprofile** package and its dependencies install all the necessary utilities for executing OProfile. To instruct OProfile to profile all the applications running on the system and to group the samples for the shared libraries with the application using the library, run the following command:

```
# opcontrol --no-vmlinux --separate=library --start
```

You can also start the OProfile daemon without collecting system data. To do so, use the option **--start-daemon**. The **--stop** option halts data collection, while **--shutdown** terminates the OProfile daemon.

Use **opreport**, **opannotate**, or **opgprof** to display the collected profiling data. By default, the data collected by the OProfile daemon is stored in `/var/lib/oprofile/samples/`.

OProfile conflict with Performance Counters for Linux (PCL) tools

Both OProfile and Performance Counters for Linux (PCL) use the same hardware Performance Monitoring Unit (PMU). If the PCL or the NMI watchdog timer are using the hardware PMU, a message like the following occurs when starting OProfile:

```
# opcontrol --start
Using default event: CPU_CLK_UNHALTED:100000:0:1:1
Error: counter 0 not available nmi_watchdog using this resource ? Try:
opcontrol --deinit
echo 0 > /proc/sys/kernel/nmi_watchdog
```

Stop any **perf** commands running on the system, then turn off the NMI watchdog and reload the OProfile kernel driver with the following commands:

```
# opcontrol --deinit
```

```
# echo 0 > /proc/sys/kernel/nmi_watchdog
```

6.2.3. OProfile in Red Hat Enterprise Linux 7

OProfile 0.9.8 has been released for Red Hat Enterprise Linux 7. This is an alpha version but has proven stable for many users. With the 0.9.8 release OProfile can now also be used to profile specific individual processes.

6.2.3.1. New Features

A new **operf** program is now available that allows non-root users to profile single processes. This can also be used for system-wide profiling, but in this case root authority is required. This capability requires a kernel version of 2.6.31 or greater.

OProfile also supports a number of new processors:

- ▶ Tiler tile64
- ▶ Tiler tilepro
- ▶ Tiler tile-gx
- ▶ IBM System z10
- ▶ IBM System z196
- ▶ Intel Ivy Bridge

- ▶ ARMv7 Cortex-A5
- ▶ ARMv7 Cortex-A15
- ▶ ARMv7 Cortex-A7

6.2.3.2. Incompatibilities with the Previous Release

OProfile 0.9.8 has some incompatibilities with the previous release:

- ▶ Support for pre-2.6 kernels has been removed.
- ▶ With the removal of pre-2.6 support, the `--with-kernel-support` configure option is no longer needed nor valid.
- ▶ Sample header mtime field has changed to u64.
- ▶ The `configure.in` file has been renamed to `configure.ac`. This should be a transparent change.

6.2.3.3. Known Problems and Limitations

OProfile 0.9.8 has a few known problems and limitations. These are:

- ▶ AMD Instruction Based Sampling (IBS) is not currently supported with the new `opperf` program. Use the **legacy** `opcontrol` commands for IBS profiling.
- ▶ When using `opperf` to profile multiple events, the absolute number of events recorded will usually be substantially fewer than expected. This is due to a bug in the Linux kernel's Performance Events Subsystem that was fixed between Linux kernel version 3.1 and 3.5.
- ▶ If NMI watchdog is not disabled on x86_64 systems, `opcontrol` may fail to allocate the hardware performance counters it needs. The progress of this issue can be followed in bugzilla at https://bugzilla.redhat.com/show_bug.cgi?id=683176.
- ▶ Many Alpha ev67 events do not work. The progress of this issue can be followed in bugzilla https://bugzilla.redhat.com/show_bug.cgi?id=931875.

6.2.4. OProfile Documentation

For a more extensive information on OProfile, refer to `man oprofile`. Red Hat Enterprise Linux also provides two comprehensive guides to OProfile in `file:///usr/share/doc/oprofile-version/`:

OProfile Manual

A comprehensive manual with detailed instructions on the setup and use of OProfile is found at `file:///usr/share/doc/oprofile-version/oprofile.html`

OProfile Internals

Documentation on the internal workings of OProfile, useful for programmers interested in contributing to the OProfile upstream, can be found at `file:///usr/share/doc/oprofile-version/internals.html`

The *OProfile Integration User Guide* in the Eclipse **Help Contents** also provides detailed information on the setup and usage of the OProfile plug-in for Eclipse. This guide is provided by the `eclipse-oprofile` package.

6.3. SystemTap

SystemTap is a useful instrumentation platform for probing running processes and kernel activity on the Linux system. To execute a probe:

1. Write *SystemTap scripts* that specify which system events (for example, virtual file system reads, packet transmissions) should trigger specified actions (for example, print, parse, or otherwise manipulate data).
2. SystemTap translates the script into a C program, which it compiles into a kernel module.
3. SystemTap loads the kernel module to perform the actual probe.

SystemTap scripts are useful for monitoring system operation and diagnosing system issues with minimal intrusion into the normal operation of the system. You can quickly instrument running system test hypotheses without having to recompile and re-install instrumented code. To compile a SystemTap script that probes *kernel-space*, SystemTap uses information from three different *kernel information packages*:

- **kernel-variant-devel-version**
- **kernel-variant-debuginfo-version**
- **kernel-debuginfo-common-arch-version**

These kernel information packages must match the kernel to be probed. In addition, to compile SystemTap scripts for multiple kernels, the kernel information packages of each kernel must also be installed.

An important new feature has been added as of Red Hat Enterprise Linux 6.1: the **--remote** option. This allows users to build the SystemTap module locally, and then execute it remotely via SSH. The syntax to use this is **--remote [USER@]HOSTNAME**; set the execution target to the specified SSH host, optionally using a different username. This option may be repeated to target multiple execution targets. Passes 1-4 are completed locally as normal to build the script, and then pass 5 copies the module to the target and runs it.

6.3.1. DynInst with SystemTap 2.0

SystemTap 2.0 introduces experimental support for running instrumentation using the *DynInst* system. DynInst is a pure-userspace binary manipulation library that allows programs to modify other running programs. It does this by inserting highly efficient instrumentation or other modifications. SystemTap 2.0 and later can use this as a backend to run a restricted class of scripts. In exchange for the restrictions, the instrumentation runs fast and entirely in user-space with no root access or kernel module operations required. The restrictions are evolving but are tighter than those for unprivileged user probing that relies on cryptography, kernel modules, and membership in special groups.

To use this experimental backend, add an extra **--runtime** option on the stap command line:

```
$ stap --runtime=stapdyn script.stp -c command
```

If the script requires facilities beyond those available with DynInst, SystemTap will advise. If this is the case, the standard kernel-module-based backends will have to be used with the **--runtime** option omitted.

6.3.2. SystemTap Compile Server

SystemTap in Red Hat Enterprise Linux 7 supports a *compile server and client* deployment. With this setup, the kernel information packages of *all* client systems in the network are installed on just one compile server host (or a few). When a client system attempts to compile a kernel module from a SystemTap script, it remotely accesses the kernel information it requires from the centralized compile server host.

A properly configured and maintained SystemTap compile server host offers the following benefits:

- ▶ The system administrator can verify the integrity of kernel information packages before making the packages available to users.
- ▶ The identity of a compile server can be authenticated using the *Secure Socket Layer (SSL)*. SSL provides an encrypted network connection that prevents eavesdropping or tampering during transmission.
- ▶ Individual users can run their own servers and authorize them for their own use as trusted.
- ▶ System administrators can authorize one or more servers on the network as trusted for use by all users.
- ▶ A server that has not been explicitly authorized is ignored, preventing any server impersonations and similar attacks.

6.3.3. SystemTap Support for Unprivileged Users

For security purposes, users in an enterprise setting are rarely given privileged (that is, root or **sudo**) access to their own machines. In addition, full SystemTap functionality should also be restricted to privileged users, as this can provide the ability to completely take control of a system.

SystemTap in Red Hat Enterprise Linux 7 features a new option to the SystemTap client: **--unprivileged**. This option allows an unprivileged user to run **stap**. Of course, several restrictions apply to unprivileged users that attempt to run **stap**.



Note

An unprivileged user is a member of the group **stapusr** but is not a member of the group **stapdev** (and is not root).

Before loading any kernel modules created by unprivileged users, SystemTap verifies the integrity of the module using standard digital (cryptographic) signing techniques. Each time the **--unprivileged** option is used, the server checks the script against the constraints imposed for unprivileged users. If the checks are successful, the server compiles the script and signs the resulting module using a self-generated certificate. When the client attempts to load the module, **staprun** first verifies the signature of the module by checking it against a database of trusted signing certificates maintained and authorized by root.

Once a signed kernel module is successfully verified, **staprun** is assured that:

- ▶ The module was created using a trusted systemtap server implementation.
- ▶ The module was compiled using the **--unprivileged** option.
- ▶ The module meets the restrictions required for use by an unprivileged user.
- ▶ The module has not been tampered with since it was created.

6.3.4. SSL and Certificate Management

SystemTap in Red Hat Enterprise Linux 7 implements authentication and security via certificates and public/private key pairs. It is the responsibility of the system administrator to add the credentials (that is, certificates) of compile servers to a database of trusted servers. SystemTap uses this database to verify the identity of a compile server that the client attempts to access. Likewise, SystemTap also uses this method to verify kernel modules created by compile servers using the **--unprivileged** option.

6.3.4.1. Authorizing Compile Servers for Connection

The first time a compile server is started on a server host, the compile server automatically generates a certificate. This certificate verifies the compile server's identity during SSL authentication and module signing.

In order for clients to access the compile server (whether on the same server host or from a client machine), the system administrator must add the compile server's certificate to a database of trusted servers. Each client host intending to use compile servers maintains such a database. This allows individual users to customize their database of trusted servers, which can include a list of compile servers authorized for their own use only.

6.3.4.2. Authorizing Compile Servers for Module Signing (for Unprivileged Users)

Unprivileged users can only load signed, authorized SystemTap kernel modules. For modules to be recognized as such, they have to be created by a compile server whose certificate appears in a database of trusted signers; this database must be maintained on each host where the module will be loaded.

6.3.4.3. Automatic Authorization

Servers started using the **stap-server** initscript are automatically authorized to receive connections from all clients on the same host.

Servers started by other means are automatically authorized to receive connections from clients on the same host run by the user who started the server. This was implemented with convenience in mind; users are automatically authorized to connect to a server they started themselves, provided that both client and server are running on the same host.

Whenever root starts a compile server, *all* clients running on the same host automatically recognize the server as authorized. However, Red Hat advises that you refrain from doing so.

Similarly, a compile server initiated through **stap-server** is automatically authorized as a trusted signer on the host in which it runs. If the compile server was initiated through other means, it is not automatically authorized as such.

6.3.5. SystemTap Documentation

For more detailed information about SystemTap, refer to the following books (also provided by Red Hat):

- ▶ *SystemTap Beginner's Guide*
- ▶ *SystemTap Tapset Reference*

The *SystemTap Beginner's Guide* and *SystemTap Tapset Reference* are also available locally when you install the **systemtap** package:

- ▶ **file:///usr/share/doc/systemtap-version/SystemTap_Beginners_Guide/index.html**
- ▶ **file:///usr/share/doc/systemtap-version/SystemTap_Beginners_Guide.pdf**
- ▶ **file:///usr/share/doc/systemtap-version/tapsets/index.html**

► <file:///usr/share/doc/systemtap-version/tapsets.pdf>

[Section 6.3.2, “SystemTap Compile Server”](#), [Section 6.3.3, “SystemTap Support for Unprivileged Users”](#), and [Section 6.3.4, “SSL and Certificate Management”](#) are all excerpts from the *SystemTap Support for Unprivileged Users and Server Client Deployment* whitepaper. This whitepaper also provides more details on each feature, along with a case study to help illustrate their application in a real-world environment.

6.4. Performance Counters for Linux (PCL) Tools and perf

Performance Counters for Linux (PCL) is a new kernel-based subsystem that provides a framework for collecting and analyzing performance data. These events will vary based on the performance monitoring hardware and the software configuration of the system. Red Hat Enterprise Linux 6 includes this kernel subsystem to collect data and the user-space tool **perf** to analyze the collected performance data.

The PCL subsystem can be used to measure hardware events, including retired instructions and processor clock cycles. It can also measure software events, including major page faults and context switches. For example, PCL counters can compute the *Instructions Per Clock* (IPC) from a process's counts of instructions retired and processor clock cycles. A low IPC ratio indicates the code makes poor use of the CPU. Other hardware events can also be used to diagnose poor CPU performance.

Performance counters can also be configured to record samples. The relative frequency of samples can be used to identify which regions of code have the greatest impact on performance.

6.4.1. Perf Tool Commands

Useful **perf** commands include the following:

perf stat

This **perf** command provides overall statistics for common performance events, including instructions executed and clock cycles consumed. Options allow selection of events other than the default measurement events.

perf record

This **perf** command records performance data into a file which can be later analyzed using **perf report**.

perf report

This **perf** command reads the performance data from a file and analyzes the recorded data.

perf list

This **perf** command lists the events available on a particular machine. These events will vary based on the performance monitoring hardware and the software configuration of the system.

Use **perf help** to obtain a complete list of **perf** commands. To retrieve **man** page information on each **perf** command, use **perf help command**.

6.4.2. Using Perf

Using the basic PCL infrastructure for collecting statistics or samples of program execution is relatively straightforward. This section provides simple examples of overall statistics and sampling.

To collect statistics on **make** and its children, use the following command:

```
# perf stat -- make all
```

The **perf** command collects a number of different hardware and software counters. It then prints the following information:

```
Performance counter stats for 'make all':

244011.782059 task-clock-msecs      #    0.925 CPUs
      53328 context-switches      #    0.000 M/sec
      515 CPU-migrations           #    0.000 M/sec
    1843121 page-faults            #    0.008 M/sec
  789702529782 cycles                # 3236.330 M/sec
1050912611378 instructions          #    1.331 IPC
  275538938708 branches             # 1129.203 M/sec
   2888756216 branch-misses        #    1.048 %
  4343060367 cache-references      #   17.799 M/sec
   428257037 cache-misses          #    1.755 M/sec

263.779192511 seconds time elapsed
```

The **perf** tool can also record samples. For example, to record data on the **make** command and its children, use:

```
# perf record -- make all
```

This prints out the file in which the samples are stored, along with the number of samples collected:

```
[ perf record: Woken up 42 times to write data ]
[ perf record: Captured and wrote 9.753 MB perf.data (~426109 samples) ]
```

As of Red Hat Enterprise Linux 6.4, a new functionality to the **{}** group syntax has been added that allows the creation of event groups based on the way they are specified on the command line.

The current **--group** or **-g** options remain the same; if it is specified for **record**, **stat**, or **top** command, all the specified events become members of a single group with the first event as a group leader.

The new **{}** group syntax allows the creation of a group like:

```
# perf record -e '{cycles, faults}' ls
```

The above results in a single event group containing *cycles* and *faults* events, with the *cycles* event as the group leader.

All groups are created with regards to threads and CPUs. As such, recording an event group within two threads on a server with four CPUs will create eight separate groups.

It is possible to use a standard event modifier for a group. This spans over all events in the group and updates each event modifier settings.

```
# perf record -r '{faults:k, cache-references}:p'
```

The above command results in the **:kp** modifier being used for *faults*, and the **:p** modifier being used for the *cache-references* event.

Performance Counters for Linux (PCL) Tools conflict with OProfile

Both OProfile and Performance Counters for Linux (PCL) use the same hardware Performance Monitoring Unit (PMU). If OProfile is currently running while attempting to use the PCL **perf** command, an error message like the following occurs when starting OProfile:

```
Error: open_counter returned with 16 (Device or resource busy). /usr/bin/dmesg may
provide additional information.

Fatal: Not all events could be opened.
```

To use the **perf** command, first shut down OProfile:

```
# opcontrol --deinit
```

You can then analyze **perf.data** to determine the relative frequency of samples. The report output includes the command, object, and function for the samples. Use **perf report** to output an analysis of **perf.data**. For example, the following command produces a report of the executable that consumes the most time:

```
# perf report --sort=comm
```

The resulting output:

```
# Samples: 1083783860000
#
# Overhead          Command
# .....
#
 48.19%           xsltproc
 44.48%           pdfxmltex
  6.01%           make
  0.95%           perl
  0.17%           kernel-doc
  0.05%           xmllint
  0.05%           cc1
  0.03%           cp
  0.01%           xmlto
  0.01%           sh
  0.01%           docproc
  0.01%           ld
  0.01%           gcc
  0.00%           rm
  0.00%           sed
  0.00%           git-diff-files
  0.00%           bash
  0.00%           git-diff-index
```

The column on the left shows the relative frequency of the samples. This output shows that **make** spends most of this time in **xsltproc** and the **pdfxmltex**. To reduce the time for the **make** to complete, focus on **xsltproc** and **pdfxmltex**. To list the functions executed by **xsltproc**, run:

```
# perf report -n --comm=xsltproc
```

This generates:

```

comm: xsltproc
# Samples: 472520675377
#
# Overhead  Samples                Shared Object  Symbol
# .....
#
  45.54%215179861044  libxml2.so.2.7.6      [.] xmlXPathCmpNodesExt
  11.63%54959620202  libxml2.so.2.7.6      [.]
xmlXPathNodeSetAdd__internal_alias
   8.60%40634845107  libxml2.so.2.7.6      [.] xmlXPathCompOpEval
   4.63%21864091080  libxml2.so.2.7.6      [.] xmlXPathReleaseObject
   2.73%12919672281  libxml2.so.2.7.6      [.]
xmlXPathNodeSetSort__internal_alias
   2.60%12271959697  libxml2.so.2.7.6      [.] valuePop
   2.41%11379910918  libxml2.so.2.7.6      [.]
xmlXPathIsNaN__internal_alias
   2.19%10340901937  libxml2.so.2.7.6      [.] valuePush__internal_alias

```

6.5. ftrace

The **ftrace** framework provides users with several tracing capabilities, accessible through an interface much simpler than SystemTap's. This framework uses a set of virtual files in the **debugfs** file system; these files enable specific tracers. The **ftrace** function tracer outputs each function called in the kernel in real time; other tracers within the **ftrace** framework can also be used to analyze wakeup latency, task switches, kernel events, and the like.

You can also add new tracers for **ftrace**, making it a flexible solution for analyzing kernel events. The **ftrace** framework is useful for debugging or analyzing latencies and performance issues that take place outside of user-space. Unlike other profilers documented in this guide, **ftrace** is a built-in feature of the kernel.

6.5.1. Using ftrace

The Red Hat Enterprise Linux kernels have been configured with the **CONFIG_FTRACE=y** option. This option provides the interfaces required by **ftrace**. To use **ftrace**, mount the **debugfs** file system as follows:

```
mount -t debugfs nodev /sys/kernel/debug
```

All the **ftrace** utilities are located in **/sys/kernel/debug/tracing/**. View the **/sys/kernel/debug/tracing/available_tracers** file to find out what tracers are available for your kernel:

```
cat /sys/kernel/debug/tracing/available_tracers
```

```
power wakeup irqsoff function sysprof sched_switch initcall nop
```

To use a specific tracer, write it to **/sys/kernel/debug/tracing/current_tracer**. For example, **wakeup** traces and records the maximum time it takes for the highest-priority task to be scheduled after the task wakes up. To use it:

```
echo wakeup > /sys/kernel/debug/tracing/current_tracer
```

To start or stop tracing, write to **/sys/kernel/debug/tracing/tracing_on**, as in:

echo 1 > /sys/kernel/debug/tracing/tracing_on (enables tracing)

echo 0 > /sys/kernel/debug/tracing/tracing_on (disables tracing)

The results of the trace can be viewed from the following files:

/sys/kernel/debug/tracing/trace

This file contains human-readable trace output.

/sys/kernel/debug/tracing/trace_pipe

This file contains the same output as **/sys/kernel/debug/tracing/trace**, but is meant to be piped into a command. Unlike **/sys/kernel/debug/tracing/trace**, reading from this file consumes its output.

6.5.2. ftrace Documentation

The **ftrace** framework is fully documented in the following files:

- ▶ *ftrace - Function Tracer*: **file:///usr/share/doc/kernel-doc-version/Documentation/trace/ftrace.txt**
- ▶ *function tracer guts*: **file:///usr/share/doc/kernel-doc-version/Documentation/trace/ftrace-design.txt**



Note

The *trace-cmd* package provides a tool of the same name that can be a useful alternative to **ftrace**. Further information is available on the **trace-cmd** man page.

Chapter 7. Red Hat Developer Toolset

Red Hat Developer Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform, and provides a complete set of development and performance analysis tools that can be installed and used on multiple versions of Red Hat Enterprise Linux. Executables built with the Red Hat Developer Toolset toolchain can then also be deployed and run on multiple versions of Red Hat Enterprise Linux.

Red Hat Developer Toolset does not replace the default system tools and libraries provided with Red Hat Enterprise Linux 6 when installed on this platform. Instead, a parallel set of developer tools provides an alternative, newer version of those tools for optional use by developers. Applications and libraries provided by Red Hat Developer Toolset do not replace the Red Hat Enterprise Linux system versions, nor are they used in preference to the system versions. The default compiler and debugger, for example, remain those provided by the base Red Hat Enterprise Linux system.

Developers can pick and choose at any time which version of the tool they would like to use by using the **sc1** utility. For details on how to install the product and invoke the executables, see the *Red Hat Developer Toolset 2.1 User Guide*.

7.1. Red Hat Developer Toolset Features

In comparison with Red Hat Enterprise Linux 6, Red Hat Developer Toolset provides more recent versions of the **Eclipse** development environment, **GNU Compiler Collection** (GCC), **GNU Debugger** (GDB), and other development and performance analysis tools as listed in [Table 7.1, “Red Hat Developer Toolset Components”](#). These updated tools allow developers to develop applications while using experimental C++11 language features, including atomic types and Transactional Memory, the latest compiler optimizations, parallel programming with OpenMP 3.1, and improved debugging support.

Table 7.1. Red Hat Developer Toolset Components

Name	Version	Description
Eclipse	4.3.1	An integrated development environment for the graphical user interface. ^[a]
GCC	4.8.2	A portable compiler suite with support for C, C++, and Fortran.
binutils	2.23.52	A collection of binary tools and other utilities to inspect and manipulate object files and binaries.
elfutils	0.157	A collection of binary tools and other utilities to inspect and manipulate ELF files.
dwz	0.11	A tool to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size.
Git	1.8.4	A distributed revision control system with a peer-to-peer architecture.
GDB	7.6.1	A command line debugger for programs written in C, C++, and Fortran.
strace	4.7	A debugging tool to monitor system calls that a program uses and signals it receives.
memstomp	0.1.4	A debugging tool to identify calls to library functions with overlapping memory regions that are not allowed by various standards.

Name	Version	Description
SystemTap	2.1	A tracing and probing tool to monitor the activities of the entire system without the need to instrument, recompile, install, and reboot.
Valgrind	3.8.1	An instrumentation framework and a number of tools to profile applications in order to detect memory errors, identify memory management problems, and report any use of improper arguments in system calls.
OProfile	0.9.8	A system-wide profiler that uses the performance monitoring hardware on the processor to retrieve information about the kernel and executables on the system.
Dyninst	8.0	A library for instrumenting and working with user-space executables during their execution.

[a] If you intend to develop applications for Red Hat JBoss Middleware or require support for OpenShift Tools, it is recommended that you use [Red Hat JBoss Developer Studio](#).

7.2. Changes in binutils

Red Hat Developer Toolset 2.0 is distributed with **binutils 2.23.52**, which provides a number of bug fixes and feature enhancements over the Red Hat Enterprise Linux system version and the version included in Red Hat Developer Toolset 1.1. Below is a comprehensive list of new features in this release.

The GNU assembler (**as**), GNU linker (**ld**), and other binary tools that are part of binutils are now released under the GNU General Public License, version 3.

7.2.1. GNU Linker

Another ELF linker, **gold**, is now available in addition to **ld**, the existing GNU linker. **gold** is intended to be a drop-in replacement for **ld**, so **ld**'s documentation is intended to be the reference documentation. **gold** supports most of **ld**'s features, except notable ones such as MRI-compatible linker scripts, cross-reference reports (**--cref**), and various other minor options. It also provides significantly improved link time with very large C++ applications.

In Red Hat Developer Toolset 2.0, the **gold** linker is not enabled by default. Users can explicitly switch between **ld** and **gold** by using the **alternatives** mechanism.

7.2.1.1. New Features

Changes Since Red Hat Enterprise Linux 6.4

The following features have been added since the release of binutils included in Red Hat Enterprise Linux 6.4:

- ▶ A new **INPUT_SECTION_FLAGS** keyword has been added to the linker script language. This keyword can be used to select input sections by section header flags.
- ▶ A new **SORT_BY_INIT_PRIORITY** keyword has been added to the linker script language. This keyword can be used to sort sections by numerical value of the GCC *init_priority* attribute encoded in the section name.
- ▶ A new **SORT_NONE** keyword has been added to the linker script language. This keyword can be used to disable section sorting.

- ▶ A new linker-provided symbol, `__ehdr_start`, has been added. When producing ELF output, this symbol points to the ELF file header (and nearby program headers) in the program's memory image.

Changes Since Red Hat Enterprise Linux 5.9

The following features have been added since the release of binutils included in Red Hat Enterprise Linux 5.9:

- ▶ GNU/Linux targets now support the **STB_GNU_UNIQUE** symbol binding, a GNU extension to the standard set of ELF symbol bindings. The binding is passed on to the dynamic linker, which ensures that in the entire process there is only one symbol with the given name and type in use.



Note

The implementation of this feature depends on capabilities only found in newer versions of the **glibc** library. Consequently, this feature is currently available in Red Hat Developer Toolset for Red Hat Enterprise Linux 6.

- ▶ A new command line option **--no-export-dynamic** has been added. This option can be used to undo the effect of the **-E** and **--export-dynamic** options.
- ▶ A new command line option **--warn-alternate-em** has been added. This option can be used to display a warning if an ELF format object file uses an alternate machine code.
- ▶ A new linker script function **REGION_ALIAS** has been added. This function can be used to create alias names of memory regions.
- ▶ A new command line option **-Ttext-segment *address*** has been added for ELF-based targets. This option can be used to set the address of the first byte of the text segment.
- ▶ A new linker script command **INSERT** has been added. This command can be used to augment the default script.
- ▶ In a linker script input section, it is now possible to specify a file within an archive by using the ***archive:file*** syntax.
- ▶ The **--sort-common** command line option now accepts **ascending** and **descending** as optional arguments. This can be used to specify which sorting order to use.
- ▶ A new command line option **--build-id** has been added for ELF-based targets. This option can be used to generate a unique per-binary identifier embedded in a note section.
- ▶ A new command line option **--default-script=*file_name*** (or **-dT *file_name***) has been added. This option can be used to specify a replacement for the built-in linker script.
- ▶ A new command line option **-Bsymbolic-functions** has been added. When creating a shared library, this option will cause references to global function symbols to be bound to the definitions with the shared library, if such exist.
- ▶ The new command line options **--dynamic-list-cpp-new** and **--dynamic-list-data** have been added, which can be used to modify the dynamic list.

7.2.1.2. Compatibility Changes

Changes Since Red Hat Enterprise Linux 6.4

The following compatibility changes have been made since the release of binutils included in Red Hat Enterprise Linux 6.4:

- ▶ The **--copy-dt-needed-entries** command line option is no longer enabled by default. Instead, **--no-copy-dt-needed-entries** is now the default option.
- ▶ Evaluation of linker script expressions has been significantly improved. Note that this can negatively affect scripts that rely on undocumented behavior of the old expression evaluation.

Changes Since Red Hat Enterprise Linux 5.9

The following compatibility changes have been made since the release of binutils included in Red Hat Enterprise Linux 5.9:

- ▶ The **--add-needed** command line option has been renamed to **--copy-dt-needed-entries** in order to avoid confusion with the **--as-needed** option.
- ▶ For GNU/Linux systems, the linker no longer processes any relocations made against symbols of the **STT_GNU_IFUNC** type. Instead, it emits them into the resulting binary for processing by the loader.



Note

The implementation of this feature depends on capabilities only found in newer versions of the **glibc** library. Consequently, this feature is currently available in Red Hat Developer Toolset for Red Hat Enterprise Linux 6.

- ▶ The **--as-needed** command line option has been adapted to link in a dynamic library in the following two cases:
 - ▶ if the dynamic library satisfies undefined symbols in regular objects, and
 - ▶ if the dynamic library satisfies undefined symbols in other dynamic libraries unless the library is already found in a **DT_NEEDED** entry of one of the libraries that are already linked.
- ▶ The **-l:file_name** command line option now searches the library path for a file name called *file_name* without adding the **.a** or **.so** file extension.

7.2.2. GNU Assembler

7.2.2.1. New Features

Changes Since Red Hat Enterprise Linux 6.4

The following features have been added since the release of binutils included in Red Hat Enterprise Linux 6.4:

- ▶ The GNU Assembler no longer requires double ampersands in macros.
- ▶ A new **--compress-debug-sections** command line option has been added to enable compression of DWARF debug information sections in the relocatable output file. Compressed debug sections are currently supported by the **readelf**, **objdump**, and **gold** tools, but not by **ld**.

- ▶ Support for `.bundle_align_mode`, `.bundle_lock`, and `.bundle_unlock` directives for x86 targets has been added..
- ▶ On x86 architectures, the GNU Assembler now allows `rep bsf`, `rep bsr`, and `rep ret` syntax.

Changes Since Red Hat Enterprise Linux 5.9

The following features have been added since the release of binutils included in Red Hat Enterprise Linux 5.9:

- ▶ GNU/Linux targets now support `gnu_unique_object` as a value of the `.type` pseudo operation. This value can be used to mark a symbol as globally unique in the entire process.
- ▶ Support for the new discriminator column in the DWARF line table with a discriminator operand for the `.loc` directive has been added.
- ▶ The `.type` pseudo operation now accepts a type of `STT_GNU_IFUNC`. This can be used to indicate that if the symbol is the target of a relocation, its value should not be used. Instead, the function should be invoked and its result used as the value.
- ▶ A new pseudo operation `.cfi_val_encoded_addr` has been added. This pseudo operation can be used to record constant addresses in unwind tables without runtime relocation.
- ▶ A new command line option `-mssse-check=[none|error|warning]` has been added for x86 targets.
- ▶ The `-a` command line option now accepts `g` as a valid sub-option. This combination can be used to enable assembly listings with additional information about the assembly, including the list of supplied command line options or the assembler version.
- ▶ A new command line option `-mssse2avx` has been added for x86 targets. This option can be used to encode SSE instructions with VEX prefix.
- ▶ x86 targets now support the Intel XSAVE, EPT, MOVBE, AES, PCLMUL, and AVX/FMA instructions.
- ▶ New command line options `-march=cpu[,+extension...]`, `-mtune=cpu`, `-mmnemonic=[att|intel]`, `-msyntax=[att|intel]`, `-mindex-reg`, `-mnaked-reg`, and `-mold-gcc` have been added for x86 targets.
- ▶ New pseudo operations `.string16`, `.string32`, and `.string64` have been added. These pseudo operations be used to generate wide character strings.
- ▶ The i386 port now supports the SSE5 instruction set.
- ▶ A new pseudo operation `.reloc` has been added. This pseudo operation serves as a low-level interface for creating relocations.

7.2.3. Other Binary Tools

7.2.3.1. New Features

Changes Since Red Hat Developer Toolset 1.1

The following features have been added since the release of binutils included in Red Hat Developer Toolset 1.1:

- ▶ A manual page for the **dwp** utility has been added.
- ▶ The binary tools now provide support for the AMD Family 15h processors, models 02h and 10-1fh.

Changes Since Red Hat Enterprise Linux 6.4

The following features have been added since the release of binutils included in Red Hat Enterprise Linux 6.4:

- ▶ The **readelf** and **objdump** tools can now display the contents of the **.debug.macro** sections.
- ▶ New **--dwarf-start** and **--dwarf-end** command line options have been added to the **readelf** and **objdump** tools. These options are used by the new Emacs mode (see the **dwarf-mode.el** file).
- ▶ A new **--interleave-width** command line option has been added to the **objcopy** tool to allow the use of the **--interleave** to copy a range of bytes from the input to the output.
- ▶ A new **--dyn-syms** command line option has been added to the **readelf** tool. This option can be used to dump dynamic symbol table.
- ▶ A new tool, **elfedit**, has been added to binutils. This tool can be used to directly manipulate ELF format binaries.
- ▶ A new command line option **--addresses** (or **-a** for short) has been added to the **addr2line** tool. This option can be used to display addresses before function and source file names.
- ▶ A new command line option **--pretty-print** (or **-p** for short) has been added to the **addr2line** tool. This option can be used to produce human-readable output.
- ▶ Support for **dwz -m** optimized debug information has been added.
- ▶ The *devtoolset-2-binutils-devel* package now provides the **demangle.h** header file.

Changes Since Red Hat Enterprise Linux 5.9

The following features have been added since the release of binutils included in Red Hat Enterprise Linux 5.9:

- ▶ A new command line option **--insn-width=width** has been added to the **objdump** tool. This option can be used to specify the number of bytes to be displayed on a single line when disassembling instructions.
- ▶ A new command line option **--relocated-dump=name|number** has been added to the **readelf** tool. This option can be used to display the relocated contents of a section as a sequence of bytes.
- ▶ A new command line option **--external-symbols-table=filename** has been added to the **gprof** tool. This option can be used to read a symbol table from a certain file.
- ▶ **bfd** now supports a plugin target, which can be used to get basic support for new file formats by having the plugin target load the same shared objects used by **gold**.
- ▶ The **--dwarf** (or **-W** for short) command line option of the **objdump** tool has been adapted to be as flexible as the **--debug-dump** (or **-w**) option of **readelf**.

- ▶ New command line options `--prefix=prefix` and `--prefix-strip=level` have been added to the `objdump` tool. These options can be used to add absolute paths for the `--source` (or `-S` for short) option.
- ▶ A new command line option `-wL` has been added to the `readelf` tool. This option can be used to dump decoded contents of the `.debug_line` section.
- ▶ “Thin” archives are now supported. Instead of containing object files, such archives contain just pathnames pointing to those files.
- ▶ A new command line option `-F` has been added to the `objdump` tool. This option can be used to include file offsets in the disassembly.
- ▶ A new command line option `-c` has been added to the `readelf` tool. This option can be used to allow string dumps of archive symbol index.
- ▶ The i386 port now supports the SSE5 instruction set.
- ▶ A new command line option `-p` has been added to the `readelf` tool. This option can be used to allow string dumps of sections.

7.2.3.2. Compatibility Changes

Changes Since Red Hat Enterprise Linux 5.9

The following compatibility changes have been made since the release of binutils included in Red Hat Enterprise Linux 5.9:

- ▶ The `--as-needed` command line option has been adapted to link in a dynamic library in the following two cases:
 - ▶ if the dynamic library satisfies undefined symbols in regular objects, and
 - ▶ if the dynamic library satisfies undefined symbols in other dynamic libraries unless the library is already found in a `DT_NEEDED` entry of one of the libraries that are already linked.

7.3. Platform Compatibility

Red Hat Developer Toolset 2.1 is available for Red Hat Enterprise Linux 5 and 6, both for 32-bit and 64-bit Intel and AMD architectures.

[Figure 7.1, “Red Hat Developer Toolset 2.1 Compatibility Matrix”](#) illustrates the support for binaries built with Red Hat Developer Toolset on a certain version of Red Hat Enterprise Linux when those binaries are run on various other versions of this system. On the y-axis, find the version of Red Hat Enterprise Linux the binary is being built on, and cross-reference with the version of Red Hat Enterprise Linux the application is being run on.



Figure 7.1. Red Hat Developer Toolset 2.1 Compatibility Matrix

7.4. Red Hat Developer Toolset References

For more information about Red Hat Developer Toolset 2.1 and what it offers, see the following resources:

- ▶ [Red Hat Developer Toolset 2.1 Release Notes](#) — The *Release Notes* for Red Hat Developer Toolset 2.1 provides important information available at the time of its release. Read this book to learn about system requirements or known problems of the product.
- ▶ [Red Hat Developer Toolset 2.1 User Guide](#) — The *User Guide* for Red Hat Developer Toolset 2.1 provides an overview of the product, explains how to invoke and use the Red Hat Developer Toolset versions of the tools, and links to resources with more in-depth information. Read this book to learn how to obtain, install, and use Red Hat Developer Toolset on your system, or for a detailed list of changes in this product.

Chapter 8. Red Hat Software Collections

For certain applications, a more recent version of some software component is often needed in order to use its latest new features. Red Hat Software Collections is a Red Hat offering that provides a set of dynamic programming languages, database servers, and related packages that are either more recent than their equivalent versions included in the base Red Hat Enterprise Linux system, or are available for this system for the first time.

Dynamic languages, database servers, and other tools distributed with Red Hat Software Collections do *not* replace the default system tools provided with Red Hat Enterprise Linux 6, nor are they used in preference to these tools. The default versions of Perl and PostgreSQL, for example, remain those provided by the base Red Hat Enterprise Linux system. Users can pick and choose at any time which version of the tool they would like to run by using the `sc1` utility.

With the notable exception of **Node.js**, all Red Hat Software Collections components are fully supported under Red Hat Enterprise Linux Subscription Level Agreements, are functionally complete, and are intended for production use.

8.1. Red Hat Software Collections Features

Red Hat Software Collections 1.0 provides recent versions of the tools listed in [Table 8.1, “Red Hat Software Collections 1.0 Components”](#).

Table 8.1. Red Hat Software Collections 1.0 Components

Component	Software Collection	Description
Perl 5.16.3	<i>perl516</i>	A recent stable release of Perl with a number of additional utilities, scripts, and <i>database connectors for MySQL and PostgreSQL</i> . This version provides a large number of new features and enhancements, including new debugging options, improved Unicode support, and better performance.
PHP 5.4.14	<i>php54</i>	A recent stable release of PHP with PEAR 1.9.4 and a number of additional utilities. This version provides new language syntax, a built-in web server for the command line, and improved performance.
Python 2.7	<i>python27</i>	A recent stable release of Python with a number of additional utilities and <i>database connectors for MySQL and PostgreSQL</i> . This version provides various new features and enhancements, including a new ordered dictionary type, faster I/O operations, and improved forward compatibility with Python 3.
Python 3.3	<i>python33</i>	A recent stable release of Python 3 with a number of additional utilities and <i>a database connector for PostgreSQL</i> . This Software Collection gives developers on Red Hat Enterprise Linux 6 access to Python 3 and allows them to test the compatibility of their applications with this version.
Ruby 1.9.3	<i>ruby193</i>	A recent stable release of Ruby with Rails 3.2.8 and <i>a large collection of Ruby gems</i> . This Software Collection gives developers on Red Hat Enterprise Linux 6 access to Ruby 1.9, which provides a number of new features and enhancements, including improved Unicode support, enhanced threading, and faster load times.

Component	Software Collection	Description
MariaDB 5.5	<i> mariadb55 </i>	A recent stable release of MariaDB. This Software Collection gives users of Red Hat Enterprise Linux 6 an alternative to MySQL, which is binary compatible with MariaDB and can be replaced with it without any data conversions.
MySQL 5.5	<i> mysql55 </i>	A recent stable release of MySQL. This version provides a number of new features and enhancements, including improved performance.
PostgreSQL 9.2	<i> postgresql92 </i>	A recent stable release of PostgreSQL. This version provides a number of new features and enhancements, including native JSON support, improved scalability, and better performance.
Node.js 0.10 ^[a]	<i> nodejs010 </i>	A recent stable release of Node.js. This Software Collection gives users of Red Hat Enterprise Linux 6 access to this programming platform.

[a] In Red Hat Software Collections 1.0, **Node.js** is included as a Technology Preview. For more information about Red Hat Technology Previews, see <http://access.redhat.com/support/offerings/techpreview/>.

8.2. Platform Compatibility

Red Hat Software Collections 1.0 is available for the following versions of the Red Hat Enterprise Linux system on AMD64 and Intel 64 architectures:

- ▶ Red Hat Enterprise Linux 6.2 Extended Update Support
- ▶ Red Hat Enterprise Linux 6.3 Extended Update Support
- ▶ Red Hat Enterprise Linux 6.4

8.3. Red Hat Software Collections Use

To run an executable from a particular Software Collection, type the following command at a shell prompt:

```
scl enable software_collection... 'command...'
```

Replace *software_collection* with a space-separated list of Software Collections you want to use and *command* with the command you want to run. For example, to execute a Perl program stored in a file named **hello.pl** with the Perl interpreter from the *perl516* Software Collection, type:

```
~]$ scl enable perl516 'perl hello.pl'  
Hello, World!
```

To start a new shell session with executables from a selected Software Collection in preference to their Red Hat Enterprise Linux equivalents, type the following at a shell prompt:

```
scl enable software_collection... bash
```

Replace *software_collection* with a space-separated list of Software Collections you want to use. For example, to start a new shell session with the *python27* and *postgresql92* Software Collections as default, type:

```
~]$ scl enable python27 postgresql92 bash
```

The list of Software Collections that are enabled in the current session is stored in the `$X_SCLS` environment variable, for instance:

```
~]$ echo $X_SCLS
python27 postgresql92
```

You can execute any command using the `scl` utility, causing it to be run with the executables from a selected Software Collection in preference to their possible Red Hat Enterprise Linux system equivalents. See [Section 4.7, “Software Collections and `scl-utils`”](#) for more information on how to install and use this command line utility on your system.

For a complete list of Software Collections that are distributed with Red Hat Software Collections, see [Table 8.1, “Red Hat Software Collections 1.0 Components”](#). For detailed instructions on how to use these Software Collections, see the *Red Hat Software Collections 1.0 Release Notes*.

8.4. Deploy Applications That Use Red Hat Software Collections

In general, you can use one of the following two approaches to deploy an application that depends on a component from Red Hat Software Collections in production:

- ▶ install all required Software Collections and packages manually and then deploy your application, or
- ▶ create a new Software Collection for your application and specify all required Software Collections and other packages as dependencies.

For more information on how to manually install individual Red Hat Software Collections components, see the *Red Hat Software Collections 1.0 Release Notes*. For a detailed explanation of how to create a custom Software Collection, read the *Red Hat Developer Toolset Software Collections Guide*.

8.5. Where to Find More Information?

- ▶ [Red Hat Software Collections 1.0 Release Notes](#) — The *Release Notes* for Red Hat Software Collections 1.0 provides important information available at the time of its release. Read this book if you want to learn about system requirements or known problems of the product.

Chapter 9. Documentation Tools

Red Hat Enterprise Linux 6 has two documentation tools available to include documentation with a project. These are Publican and Doxygen.

9.1. Publican

Publican a program is used to publish and process documentation through DocBook XML. In the process of publishing books, it checks the XML to ensure it is valid and in a publishable standard. It is particularly useful for publishing the documentation accompanying a newly created application.

9.1.1. Commands

Publican has a vast number of commands and actions available, all of which can be found in the `--help` or `--man` pages. The most common ones are:

build

Converts the XML files into other formats more suitable for documentation (PDF, HTML, HTML-single, for example).

create

Creates a new book, including all the required files as discussed in [Section 9.1.3, “Files”](#).

create_brand

Creates a new brand, allowing all books to look the same, as discussed in [Section 9.1.6, “Brands”](#).

package

Packages the files of a book into an RPM ready to distribute.

9.1.2. Create a New Document

Use the **publican create** command to create a new document including all the required files.

There are a number of options available to append to the **publican create**. These are:

--help

Prints a list of accepted options for the **publican create** command.

--name *Doc_Name*

Set the name of the book. Keep in mind that the title must contain no spaces.

--lang *Language_Code*

If this is not set, the default is en-US. The **--lang** option sets the **xml_lang** in the **publican.cfg** file and creates a directory with this name in the document directory.

--version *version*

Set the version number of the product the book is about.

--product *Product_Name*

Set the name of the product the book is about. Keep in mind that this must contain no spaces.

--brand *brand*

Set the name of a brand to use to keep the look of the documents consistent.

Refer to **--help** for more options.

Remember to change into the directory the book is to be created in before running **publican create**. This prevents the files and directories be added to the user's home directory.

9.1.3. Files

When a book is made, a number of files are created in the book's directory. These files are required for the book to be built properly and should not be deleted. They are, however, required to be edited for links (such as chapters) to work, as well as to contain the correct information regarding authors and titles etc. These files are:

publican.cfg

This file configures the build options and always includes the parameters **xml_lang** (the language the book is in, en-US for example), **type** (the type of document, a book or a set, for example), and **brand** (the branding the document uses, found here: [Section 9.1.6, "Brands"](#). Red Hat, for example.). There are a number of optional parameters but these should be used cautiously as they can cause problems further on in areas like translation. A full list of these advanced parameters can be found in the Publican User Guide. The **publican.cfg** file is unlikely to be edited much beyond the initial creation.

book_info.xml

This file is the template of the book. It contains information such as the title, subtitle, author, publication number, and the book's ID number. It also contains the basic Publican information printed at the beginning of each publication with information on the notes, cautions, and warnings as well as a basic stylistic guide. This file will be edited often as every time a book is updated the publication number has to be incremented.

Author_Group.xml

This file is used to store information about the authors and contributors. Once initially set up it is unlikely further editing will be required unless a change of authorship occurs.

Chapter.xml

This file is an example of what the actual content will be. It is created as a place holder but unless it is linked in the **Doc_Name.xml** (below) it will not appear in the actual book. When writing content for the publication, new XML files are created, named appropriately (ch-publican.xml, for example) and linked in **Doc_Name.xml**. When the book is built, the content of this file will form the content of the book. This specific file is unlikely to ever be edited but others like it will be edited constantly as content is changed, updated, added to or removed.

Doc_Name.xml

This file is the contents page of the publication. It contains a list of links to the various chapters a book is to contain. It will not actually be called 'Doc_Name' but will have whatever the title of the publication is in it's place (Developer_Guide.xml, for example). This will only be edited when new

chapters are added, removed or rearranged. This must remain the same as **Doc_Name.ent** or the book will not build.

Doc_Name.ent

This file contains a list of local entities. By default **YEAR** is set to the current year and **HOLDER** has a reminder to place the copyright owner's name there. As with **Doc_Name.xml**, this file will not be called 'Doc_Name' but will be replaced with the title of the document (**Developer_Guide.ent**, for example). This is only likely to be edited once at the beginning of publication or if the copyright owner changes. This must remain the same as **Doc_Name.xml** or the book will not build.

Revision_History.xml

When **publican package** is run, the first XML file containing a **<revhistory>** tag is used to build the RPM revision history.

9.1.3.1. Adding Media to Documentation

Occasionally it may become necessary to add various media to a document in order to illustrate what is being explained.

Images

The **images** folder is created by publican in the document's directory. Store any images used in the document here. Then when entering an image into the document, link to the image inside the **images** directory (**./images/image1.png**, for example).

Code Examples

As time passes and technology changes, a project's documentation will be required to be updated to reflect differences in code. To make this easier, create individual files for each code example in a preferred editor, then save them in a folder called **extras** in the document's directory. Then, when entering the code sample into the document, link to the file and the folder it is in. This way an example used in several places can be updated only once, and rather than search through a document looking for a specific item to change, all the code examples are located in the one place, saving time and effort.

Arbitrary Files

On occasion there may be a requirement for files not attached to the documentation to be bundled with the RPM (video tutorials, for example). Adding these files to a directory called **files** in the publication's directory will allow them to be added to the RPM when the book is compiled.

To link to any of these files, use the following XML:

```
<xi:include parse="text" href="extras/fork/fork1.c"
xmlns:xi="http://www.w3.org/2001/XInclude" />
```

9.1.4. Building a Document

In the root directory, first run a test build to ensure that all the XML is correct and acceptable by typing **publican build --formats=chosen_format --langs=chosen_language**. For example, to build a document in US English and as a single HTML page, run **publican build --formats=html-single --langs=en-US**. Provided there are no errors the book will be built into the root directory where the pages can be viewed to see if it has the look required. It is recommended to do this regularly in order to make troubleshooting as easy as possible.



Note

When creating a build to test for any bugs in the XML code, sometimes it may be useful to use the `-novalid` option. This skips over any cross-references and links that point to files or sections of the document that do not yet exist. Instead they are shown as three question marks (???)

There are a number of different formats a document can be published in. These are:

html

An ordinary HTML page with links to new pages for new chapters and sections.

html-single

One long HTML page where the links to new chapters and sections at the top of the page directing the user further down the page, rather than to new page.

html-desktop

One long HTML page where the links to new chapters and sections are in a panel on the left side of the document, directing the user further down the page, rather than to a new page.

man

A man page for Linux, UNIX, and other similar operating systems.

pdf

A PDF file.

test

The XML is validated without actually creating a file for viewing.

txt

A single text file.

epub

An e-book in EPUB format.

eclipse

An Eclipse help plug-in.

9.1.5. Packaging a Publication

Once the documentation is complete and can be built with no errors, run `publican package --lang=chosen_language`. This will output SRPM packages to `tmp/rpm` in the document's directory, and binary RPM packages will go to `tmp/rpm/noarch` in the document's directory. By default, these packages are named `productname-title-productnumber-[web]-language-edition-pubsnumber.[build_target].noarch.file_extension` with the information for each of these sections coming from `publican.cfg`.

9.1.6. Brands

Brands are used in a similar way as templates in that they create a level of consistency in appearance, with aspects like matching logos, images and color schemes, across a range of documents. This can be particularly useful when producing several books for the same application or the same bundle of applications.

In order to create a new brand, it must have a name and a language. Run **publican create_brand --name=brand --lang=language_code**. This will create a folder called **publican-brand** and place it in the publication's directory. This folder contains the following files:

COPYING

Part of an SRPM package and containing the copyright license and details.

defaults.cfg

Provides default values for the parameters that can be set in **publican.cfg**. Specifications from this file are applied first before applying those in the **publican.cfg** file. Therefore, values in the **publican.cfg** file override those in the **defaults.cfg** file. It is best used for aspects that are routinely used throughout the documents but still allows writers to change settings.

overrides.cfg

Also provides values for the parameters that can be set in **publican-brand.spec**. Specifications from this file are applied last, thus overriding both the **defaults.cfg** and the **publican.cfg**. It is best used for aspects the writers are not allowed to change.

publican.cfg

This file is similar to the **publican.cfg** file for a publication in that it configures basic information for the brand, such as version, release number and brand name.

publican-brand.spec

This file is used by the RPM Package Manager to package the publication into an RPM.

README

Part of an SRPM package and providing a brief description of the package.

A subdirectory, named by the language code, is also placed in this directory and contains the following files:

Feedback.xml

This is generated by default to allow readers to leave feedback. Customize it to contain the relevant contact details or a bug reporting process.

Legal_Notice.xml:

Contains copyright information. Edit it to change the details of the chosen copyright license.

Two more subdirectories are within this directory. The **images** subdirectory contains a number of images of both raster (PNG) and vector (SVG) formats and serve as place holders for various navigation icons that can be changed by replacing the images. The **css** folder contains **overrides.css**, which sets the visual style for the brand, overriding those in **common.css**.

In order to package the new brand ready for distribution, use the **publican package** command. By default this creates *source RPM packages* (SRPM Packages) but it can also create *binary RPM packages* using the option **--binary**. Packages are named **publican-brand-version-release.[build_target].[noarch].file_extension** with the required parameters taken from the **publican.cfg** file.



Note

SRPM packages have the file extension `.src.rpm` while binary RPM packages have the file extension `.rpm`

Binary RPM packages include **[build_target].noarch** before the file extension, where **[build_target]** represents the operating system and version that the package is built for as set by the **os_ver** parameter in the **publican.cfg** file. The **noarch** element specifies that the package can be installed on any system, regardless of the system architecture.

9.1.7. Building a Website

Publican can also build websites to manage documentation. This is mostly useful when only one person is maintaining the documentation, but where a team is working on the documentation Publican can generate RPM packages of documentation to install on a web server. The website created consists of a homepage, product and version description pages, and the pages for the documentation. In the publication's root directory, Publican creates a configuration file, an SQLite database file, and two subdirectories. There could be many configuration files depending on how many languages the documentation is published in, with a new subdirectory for each language.

Refer to [Section 9.1.8, “Documentation”](#) for more information.

9.1.8. Documentation

Publican has comprehensive **--man**, **--help** and **--help_actions** pages accessed from the terminal.

For information on XML including the different tags available, see the DocBook guide, *DocBook: the definitive guide* by Norman Walsh and Leonard Muellner, found here:

<http://www.docbook.org/tdg/en/html/docbook> and specifically [Part II: Reference](#) for a list of all the tags and brief instructions on how to use them.

There is also the comprehensive Publican User Guide accessed online at <http://jfearn.fedorapeople.org/en-US/index.html> or installed locally with **yum install publican-doc**.

9.2. Doxygen

Doxygen is a documentation tool that creates reference material both online in HTML and offline in Latex. It does this from a set of documented source files which makes it easy to keep the documentation consistent and correct with the source code.

9.2.1. Doxygen Supported Output and Languages

Doxygen has support for output in:

- ▶ RTF (MS Word)

- ▶ PostScript
- ▶ Hyperlinked PDF
- ▶ Compressed HTML
- ▶ Unix man pages

Doxygen supports the following programming languages:

- ▶ C
- ▶ C++
- ▶ C#
- ▶ Objective -C
- ▶ IDL
- ▶ Java
- ▶ VHDL
- ▶ PHP
- ▶ Python
- ▶ Fortran
- ▶ D

9.2.2. Getting Started

Doxygen uses a configuration file to determine its settings, therefore it is paramount that this be created correctly. Each project requires its own configuration file. The most painless way to create the configuration file is with the command **doxygen -g *config-file***. This creates a template configuration file that can be easily edited. The variable *config-file* is the name of the configuration file. If it is committed from the command it is called Doxyfile by default. Another useful option while creating the configuration file is the use of a minus sign (-) as the file name. This is useful for scripting as it will cause Doxygen to attempt to read the configuration file from standard input (**stdin**).

The configuration file consists of a number of variables and tags, similar to a simple Makefile. For example:

```
TAGNAME = VALUE1 VALUE2...
```

For the most part these can be left alone but should it be required to edit them refer to the [configuration page](#) of the Doxygen documentation website for an extensive explanation of all the tags available. There is also a GUI interface called **doxywizard**. If this is the preferred method of editing then documentation for this function can be found on the [Doxywizard usage page](#) of the Doxygen documentation website.

There are eight tags that are useful to become familiar with.

INPUT

For small projects consisting mainly of C or C++ source and header files it is not required to change anything. However, if the project is large and consists of a source directory or tree, then assign the root directory or directories to the INPUT tag.

FILE_PATTERNS

File patterns (for example, `*.cpp` or `*.h`) can be added to this tag allowing only files that match one of the patterns to be parsed.

RECURSIVE

Setting this to **yes** will allow recursive parsing of a source tree.

EXCLUDE and EXCLUDE_PATTERNS

These are used to further fine-tune the files that are parsed by adding file patterns to avoid. For example, to omit all **test** directories from a source tree, use **EXCLUDE_PATTERNS = */test/***.

EXTRACT_ALL

When this is set to **yes**, doxygen will pretend that everything in the source files is documented to give an idea of how a fully documented project would look. However, warnings regarding undocumented members will not be generated in this mode; set it back to **no** when finished to correct this.

SOURCE_BROWSER and INLINE_SOURCES

By setting the **SOURCE_BROWSER** tag to **yes** doxygen will generate a cross-reference to analyze a piece of software's definition in its source files with the documentation existing about it. These sources can also be included in the documentation by setting **INLINE_SOURCES** to **yes**.

9.2.3. Running Doxygen

Running **doxygen config-file** creates **html**, **rtf**, **latex**, **xml**, and / or **man** directories in whichever directory doxygen is started in, containing the documentation for the corresponding filetype.

HTML OUTPUT

This documentation can be viewed with a HTML browser that supports cascading style sheets (CSS), as well as DHTML and Javascript for some sections. Point the browser (for example, Mozilla, Safari, Konqueror, or Internet Explorer 6) to the **index.html** in the **html** directory.

LaTeX OUTPUT

Doxygen writes a **Makefile** into the **latex** directory in order to make it easy to first compile the Latex documentation. To do this, use a recent teTeX distribution. What is contained in this directory depends on whether the **USE_PDFLATEX** is set to **no**. Where this is true, typing **make** while in the **latex** directory generates **refman.dvi**. This can then be viewed with **xdvi** or converted to **refman.ps** by typing **make ps**. Note that this requires **dvips**.

There are a number of commands that may be useful. The command **make ps_2on1** prints two pages on one physical page. It is also possible to convert to a PDF if a ghostscript interpreter is installed by using the command **make pdf**. Another valid command is **make pdf_2on1**. When doing this set **PDF_HYPERLINKS** and **USE_PDFLATEX** tags to **yes** as this will cause **Makefile** will only contain a target to build **refman.pdf** directly.

RTF OUTPUT

This file is designed to import into Microsoft Word by combining the RTF output into a single file: **refman.rtf**. Some information is encoded using fields but this can be shown by selecting all (**CTRL+A** or Edit -> select all) and then right-click and select the **toggle fields** option from the drop down menu.

XML OUTPUT

The output into the **xml** directory consists of a number of files, each compound gathered by doxygen, as well as an **index.xml**. An XSLT script, **combine.xslt**, is also created that is used to combine all the XML files into a single file. Along with this, two XML schema files are created, **index.xsd** for the index file, and **compound.xsd** for the compound files, which describe the possible elements, their attributes, and how they are structured.

MAN PAGE OUTPUT

The documentation from the **man** directory can be viewed with the **man** program after ensuring the **manpath** has the correct man directory in the man path. Be aware that due to limitations with the man page format, information such as diagrams, cross-references and formulas will be lost.

9.2.4. Documenting the Sources

There are three main steps to document the sources.

1. First, ensure that **EXTRACT_ALL** is set to **no** so warnings are correctly generated and documentation is built properly. This allows doxygen to create documentation for documented members, files, classes and namespaces.
2. There are two ways this documentation can be created:

A *special* documentation block

This comment block, containing additional marking so Doxygen knows it is part of the documentation, is in either C or C++. It consists of a brief description, or a detailed description. Both of these are optional. What is not optional, however, is the *in body* description. This then links together all the comment blocks found in the body of the method or function.



Note

While more than one brief or detailed description is allowed, this is not recommended as the order is not specified.

The following will detail the ways in which a comment block can be marked as a detailed description:

- C-style comment block, starting with two asterisks (*) in the JavaDoc style.

```
/**
 * ... documentation ...
 */
```

- C-style comment block using the Qt style, consisting of an exclamation mark (!) instead of an extra asterisk.

```
/* !
 * ... documentation ...
 */
```

- ▶ The beginning asterisks on the documentation lines can be left out in both cases if that is preferred.
- ▶ A blank beginning and end line in C++ also acceptable, with either three forward slashes or two forward slashes and an exclamation mark.

```
///
/// ... documentation
///
```

or

```
//!
//! ... documentation ...
//!
```

- ▶ Alternatively, in order to make the comment blocks more visible a line of asterisks or forward slashes can be used.

```
////////////////////////////////////
/// ... documentation ...
////////////////////////////////////
```

or

```
/******//**
 * ... documentation ...
 *****/
```

Note that the two forwards slashes at the end of the normal comment block start a special comment block.

There are three ways to add a brief description to documentation.

- ▶ To add a brief description use `\brief` above one of the comment blocks. This brief section ends at the end of the paragraph and any further paragraphs are the detailed descriptions.

```
/*! \brief brief documentation.
 *     brief documentation.
 *
 * detailed documentation.
 */
```

- ▶ By setting `JAVADOC_AUTOBRIEF` to `yes`, the brief description will only last until the first dot followed by a space or new line. Consequentially limiting the brief description to a single sentence.

```
/** Brief documentation. Detailed documentation continues * from
 here.
 */
```

This can also be used with the above mentioned three-slash comment blocks (*///*).

- The third option is to use a special C++ style comment, ensuring this does not span more than one line.

```
/// Brief documentation.
/** Detailed documentation. */
```

or

```
//! Brief documentation.
  
//! Detailed documentation //! starts here
```

The blank line in the above example is required to separate the brief description and the detailed description, and **JAVADOC_AUTOBRIEF** must to be set to **no**.

Examples of how a documented piece of C++ code using the Qt style can be found on the [Doxygen documentation website](#)

It is also possible to have the documentation after members of a file, struct, union, class, or enum. To do this add a < marker in the comment block.\

```
int var; /*!< detailed description after the member */
```

Or in a Qt style as:

```
int var; /**< detailed description after the member */
```

or

```
int var; //!< detailed description after the member
  
//!<
```

or

```
int var; ///< detailed description after the member
  
///<
```

For brief descriptions after a member use:

```
int var; //!< brief description after the member
```

or

```
int var; ///< brief description after the member
```

Examples of these and how the HTML is produced can be viewed on the [Doxygen documentation website](#)

Documentation at other places

While it is preferable to place documentation in front of the code it is documenting, at times it is only possible to put it in a different location, especially if a file is to be documented; after all it is impossible to place the documentation in front of a file. This is best avoided unless it is absolutely necessary as it can lead to some duplication of information.

To do this it is important to have a structural command inside the documentation block. Structural commands start with a backslash (\) or an at-sign (@) for JavaDoc and are followed by one or more parameters.

```

/*! \class Test
    \brief A test class.

    A more detailed description of class.
*/

```

In the above example the command `\class` is used. This indicates that the comment block contains documentation for the class 'Test'. Others are:

- ▶ `\struct`: document a C-struct
- ▶ `\union`: document a union
- ▶ `\enum`: document an enumeration type
- ▶ `\fn`: document a function
- ▶ `\var`: document a variable, typedef, or enum value
- ▶ `\def`: document a #define
- ▶ `\typedef`: document a type definition
- ▶ `\file`: document a file
- ▶ `\namespace`: document a namespace
- ▶ `\package`: document a Java package
- ▶ `\interface`: document an IDL interface

3. Next, the contents of a special documentation block is parsed before being written to the HTML and / Latex output directories. This includes:
 - a. Special commands are executed.
 - b. Any white space and asterisks (*) are removed.
 - c. Blank lines are taken as new paragraphs.
 - d. Words are linked to their corresponding documentation. Where the word is preceded by a percent sign (%) the percent sign is removed and the word remains.
 - e. Where certain patterns are found in the text, links to members are created. Examples of this can be found on the [automatic link generation page](#) on the Doxygen documentation website.
 - f. When the documentation is for Latex, HTML tags are interpreted and converted to Latex equivalents. A list of supported HTML tags can be found on the [HTML commands page](#) on the Doxygen documentation website.

9.2.5. Resources

More information can be found on the Doxygen website.

- » [*Doxygen homepage*](#)
- » [*Doxygen introduction*](#)
- » [*Doxygen documentation*](#)
- » [*Output formats*](#)

Appendix

A.1. `malloc`

`malloc` is a library call that allows a program to change the behavior of the `malloc` memory allocator.

Example A.1. Allocator heuristics

An allocator has heuristics to determine long versus short lived objects. For the former, it attempts to allocate with `mmap`. For the later, it attempts to allocate with `sbrk`.

In order to override these heuristics, set `M_MMAP_THRESHOLD`.

In multi-threaded applications, the allocator creates multiple *arenas* in response to lock contention in existing arenas. This can improve the performance significantly for some multi-threaded applications at the cost of an increase in memory usage. To keep this under control, limit the number of arenas that can be created by using the `malloc` interface.

The allocator has limits on the number of arenas it can create. For 32bit targets, it will create $2 * \#$ core arenas; for 64bit targets, it will create $8 * \#$ core arenas. `malloc` allows the developer to override those limits.

Example A.2. `malloc`

To ensure no more than eight arenas are created, issue the following library call:

```
malloc (M_ARENA_MAX, 8);
```

The first argument for `malloc` can be:

- ▶ `M_MXFAST`
- ▶ `M_TRIM_THRESHOLD`
- ▶ `M_TOP_PAD`
- ▶ `M_MMAP_THRESHOLD`
- ▶ `M_MMAP_MAX`
- ▶ `M_CHECK_ACTION`
- ▶ `M_PERTURB`
- ▶ `M_ARENA_TEST`
- ▶ `M_ARENA_MAX`

Specific definitions for the above can be found at <http://www.makelinux.net/man/3/M/mallopt>.

`malloc_trim`

`malloc_trim` is a library call that requests the allocator return any unused memory back to the

operating system. This is normally automatic when an object is freed. However, in some cases when freeing small objects, **glibc** might not immediately release the memory back to the operating system. It does this so that the free memory can be used to satisfy upcoming memory allocation requests as it is expensive to allocate from and release memory back to the operating system.

malloc_stats

malloc_stats is used to dump information about the allocator's internal state to **stderr**. Using **mallinfo** is similar to this, but it places the state into a structure instead.

Further Information

More information on **mallopt** can be found at <http://www.makelinux.net/man/3/M/mallopt> and http://www.gnu.org/software/libc/manual/html_node/Malloc-Tunable-Parameters.html.

Revision History

Revision 1-1	Mon Jun 2 2014	Jacquelynn East
Version for 7.0 GA release		
Revision 0-33	Wed May 21 2014	Jacquelynn East
Added url to bug in oprofile section BZ#795987		
Revision 0-32	Mon Apr 12 2014	Jacquelynn East
Rebuild for style changes		
Revision 0-21	Tue Apr 8 2014	Jacquelynn East
New features added to the java section BZ#795590		
Important note in btrfs has been edited to reflect tech preview status BZ#1025572		
Revision 0-20	Mon Apr 7 2014	Jacquelynn East
Added sentence stating RHEL7 constantly updated for latest JDK version BZ#911433		
Updated SystemTap chapter BZ#795980		
Updated GCC and DTS sections BZ#1032615		
Added GCC power defaults BZ#1009827		
Revision 0-18	Fri Apr 4 2014	Laura Bailey
Added operf command to OProfile section and made minor corrections for BZ#1048636.		
Revision 0-17	Thu Apr 3 2014	Jacquelynn East
Added section on writing plug-ins for the libStorageMgmt library BZ#795222		
Revision 0-16	Fri Mar 14 2014	Jacquelynn East
Updated the DTS chapter BZ#1026417		
Revision 0-15	Mon Feb 3 2014	Jacquelynn East
Several minor edits to perl section BZ#795583		
Revision 0-13	Fri Jan 10 2014	Jacquelynn East
Updated oProfile section BZ#795987		
Revision 0-12	Wed Nov 27 2013	Jacquelynn East
Updated python section to Python 2.7 BZ#795585		
Updated perl section to Perl 5.16 BZ#795583		
Updated ruby section to Ruby 1.9 BZ#795629 and BZ#795631		
Revision 0-9	Wed Nov 27 2013	Jacquelynn East
Pre beta build		
Revision 0-7	Wed Oct 2 2013	Jacquelynn East
Changed references to /bin, /sbin, /lib, /lib64 to be nested under /usr BZ#952479		
Revision 0-6	Mon Sep 23 2013	Jacquelynn East
Added important note regarding a possible memcpy failure BZ#836490		
Revision 0-5	Wed Sep 11 2013	Jacquelynn East

Branch for Red Hat Enterprise Linux 7.0.

Index

Symbols

.spec file

- specfile Editor
- compiling and building, [Eclipse RPM Building](#), [Eclipse Built-in Specfile Editor](#)

A

ABI

- compatibility, [ABI Compatibility](#)

addr2line

- features, [New Features](#)

advantages

- Python pretty-printers
- debugging, [Python Pretty-Printers](#)

Akonadi

- KDE Development Framework
- libraries and runtime support, [KDE4 Architecture](#)

Apache Subversion (SVN)

- Collaborating, [Apache Subversion \(SVN\)](#)
 - Committing changes, [Committing Changes](#)
 - Documentation, [SVN Documentation](#)
 - Importing data, [Importing Data](#)
 - Installation, [Installation](#)
 - SVN Repository, [SVN Repository](#)
 - Working Copies, [Working Copies](#)

application binary interface (see ABI)

architecture, KDE4

- KDE Development Framework
- libraries and runtime support, [KDE4 Architecture](#)

authorizing compile servers for connection

- SSL and certificate management
- SystemTap, [Authorizing Compile Servers for Connection](#)

automatic authorization

- SSL and certificate management
- SystemTap, [Automatic Authorization](#)

Autotools

- compiling and building, [Autotools](#)

B

backtrace

- tools
 - GNU debugger, [Simple GDB](#)

bfd

- features, [New Features](#)

binutils

- version, [Red Hat Developer Toolset Features](#)

Boost

- libraries and runtime support, [Boost](#)

boost-doc

- Boost
 - libraries and runtime support, [Boost Documentation](#)

breakpoint

- fundamentals
 - GNU debugger, [Simple GDB](#)

breakpoints (conditional)

- GNU debugger, [Conditional Breakpoints](#)

build-id

- compiling and building, [build-id Unique Identification of Binaries](#)

building

- compiling and building, [Compiling and Building](#)

C**C++ Standard Library, GNU**

- libraries and runtime support, [The GNU C++ Standard Library](#)

C++0x, added support for

- GNU C++ Standard Library
 - libraries and runtime support, [GNU C++ Standard Library Updates](#)

C++11 (see GNU Compiler Collection)**C/C++ source code**

- Eclipse, [Editing C/C++ Source Code in Eclipse](#)

C11 (see GNU Compiler Collection)**cachegrind**

- tools
 - Valgrind, [Valgrind Tools](#)

callgrind

- tools
 - Valgrind, [Valgrind Tools](#)

CDT in Eclipse

- Compiling and building, [CDT in Eclipse](#)
- Compiling and Building
 - Autotools Project, [Autotools Project](#)
 - Managed Make Project, [Managed Make Project](#)
 - Standard Make Project, [Standard Make Project](#)

certificate management

- SSL and certificate management
 - SystemTap, [SSL and Certificate Management](#)

Code Completion

- libhover
 - libraries and runtime support, [Setup and Usage](#)

Collaborating, [Collaborating](#)

- Apache Subversion (SVN), [Apache Subversion \(SVN\)](#)
 - Committing changes, [Committing Changes](#)
 - Documentation, [SVN Documentation](#)
 - Importing data, [Importing Data](#)
 - Installation, [Installation](#)
 - SVN Repository, [SVN Repository](#)
 - Working Copies, [Working Copies](#)
- Concurrent Versions System (CVS), [Concurrent Versions System \(CVS\)](#)

Command Group Availability Tab

- integrated development environment
 - Eclipse, [Customize Perspective](#)

commands

- fundamentals
 - GNU debugger, [Simple GDB](#)
- profiling
 - Valgrind, [Valgrind Tools](#)
- tools
 - Performance Counters for Linux (PCL) and perf, [Perf Tool Commands](#)

commonly-used commands

- Autotools
 - compiling and building, [Autotools](#)

compat-glibc

- libraries and runtime support, [compat-glibc](#)

compatibility

- GNU Compiler Collection, [Language Compatibility](#), [Compatibility Changes](#), [Fortran 2003 Compatibility](#), [Fortran 2008 Compatibility](#), [Fortran 77 Compatibility](#), [ABI Compatibility](#), [Debugging Compatibility](#), [Other Compatibility](#)
- libraries and runtime support, [Compatibility](#)

compile server

- SystemTap, [SystemTap Compile Server](#)

compiling and building

- Autotools, [Autotools](#)
 - commonly-used commands, [Autotools](#)
 - configuration script, [Configuration Script](#)
 - documentation, [Autotools Documentation](#)
 - plug-in for Eclipse, [Autotools Plug-in for Eclipse](#)
 - templates (supported), [Autotools Plug-in for Eclipse](#)
- build-id, [build-id Unique Identification of Binaries](#)
- distributed compiling, [Distributed Compiling](#)
- GNU Compiler Collection, [GNU Compiler Collection \(GCC\)](#)
- introduction, [Compiling and Building](#)
- required packages, [Distributed Compiling](#)
- specfile Editor, [Eclipse RPM Building](#), [Eclipse Built-in Specfile Editor](#)
 - plug-in for Eclipse, [Eclipse RPM Building](#), [Eclipse Built-in Specfile Editor](#)

Compiling and building

- CDT in Eclipse, [CDT in Eclipse](#)

Compiling and Building

- CDT in Eclipse
 - Autotools Project, [Autotools Project](#)
 - Managed Make Project, [Managed Make Project](#)
 - Standard Make Project, [Standard Make Project](#)

Concurrent Versions System (CVS)

- Collaborating, [Concurrent Versions System \(CVS\)](#)

conditional breakpoints

- GNU debugger, [Conditional Breakpoints](#)

configuration script

- Autotools
 - compiling and building, [Configuration Script](#)

configuring keyboard shortcuts

- integrated development environment
 - Eclipse, [Keyboard Shortcuts](#)

connection authorization (compile servers)

- SSL and certificate management
 - SystemTap, [Authorizing Compile Servers for Connection](#)

Console View

- user interface
 - Eclipse, [Eclipse User Interface](#)

Contents (Help Contents)

- Help system
 - Eclipse, [Eclipse Documentation](#)

continue

- tools
 - GNU debugger, [Simple GDB](#)

Customize Perspective Menu

- integrated development environment
 - Eclipse, [Customize Perspective](#)

D**debugfs file system**

- profiling
 - ftrace, [ftrace](#)

debugging

- debuginfo-packages, [Installing Debuginfo Packages](#)
 - installation, [Installing Debuginfo Packages](#)
- GNU debugger, [GDB](#)
 - fundamental mechanisms, [GDB](#)
 - GDB, [GDB](#)
 - requirements, [GDB](#)
- introduction, [Debugging](#)
- Python pretty-printers, [Python Pretty-Printers](#)
 - advantages, [Python Pretty-Printers](#)
 - debugging output (formatted), [Python Pretty-Printers](#)
 - documentation, [Python Pretty-Printers](#)
 - pretty-printers, [Python Pretty-Printers](#)
- variable tracking at assignments (VTA), [Variable Tracking at Assignments](#)

Debugging

- Debugging C/C++ applications with Eclipse, [Debugging C/C++ Applications with Eclipse](#)

debugging a Hello World program

- usage
 - GNU debugger, [Running GDB](#)

Debugging C/C++ applications with Eclipse

- Debugging, [Debugging C/C++ Applications with Eclipse](#)

debugging output (formatted)

- Python pretty-printers
 - debugging, [Python Pretty-Printers](#)

debuginfo-packages

- debugging, [Installing Debuginfo Packages](#)

default

- user interface
 - Eclipse, [Eclipse User Interface](#)

distributed compiling

- compiling and building, [Distributed Compiling](#)

documentation

- Autotools
 - compiling and building, [Autotools Documentation](#)
- Boost
 - libraries and runtime support, [Boost Documentation](#)
- GNU C++ Standard Library
 - libraries and runtime support, [GNU C++ Standard Library Documentation](#)
- GNU debugger, [GDB Documentation](#)
- Java
 - libraries and runtime support, [Java Documentation](#)
- KDE Development Framework
 - libraries and runtime support, [kdelibs Documentation](#)
- OProfile
 - profiling, [OProfile Documentation](#)
- Perl
 - libraries and runtime support, [Perl Documentation](#)
- profiling
 - ftrace, [ftrace Documentation](#)
- Python
 - libraries and runtime support, [Python Documentation](#)
- Python pretty-printers
 - debugging, [Python Pretty-Printers](#)
- Qt
 - libraries and runtime support, [Qt Library Documentation](#)
- Ruby
 - libraries and runtime support, [Ruby Documentation](#)
- SystemTap
 - profiling, [SystemTap Documentation](#)
- Valgrind
 - profiling, [Valgrind Documentation](#)

Documentation

- Doxygen, [Doxygen](#)
 - Document sources, [Documenting the Sources](#)
 - Getting Started, [Getting Started](#)
 - Resources, [Resources](#)
 - Running Doxygen, [Running Doxygen](#)
 - Supported output and languages, [Doxygen Supported Output and Languages](#)

Documentation Tools, [Documentation Tools](#)

- Publican, [Publican](#)
 - Adding media to documentation, [Adding Media to Documentation](#)

- Brands, [Brands](#)
- Building a document, [Building a Document](#)
- Building a website, [Building a Website](#)
- Commands, [Commands](#)
- Create a new document, [Create a New Document](#)
- Files, [Files](#)
- Packaging a publication, [Packaging a Publication](#)
- Publican documentation, [Documentation](#)

Doxygen

- Documentation, [Doxygen](#)
 - document sources, [Documenting the Sources](#)
 - Getting Started, [Getting Started](#)
 - Resources, [Resources](#)
 - Running Doxygen, [Running Doxygen](#)
 - Supported output and languages, [Doxygen Supported Output and Languages](#)

dwz

- version, [Red Hat Developer Toolset Features](#)

Dynamic Help

- Help system
 - Eclipse, [Eclipse Documentation](#)

Dyninst

- version, [Red Hat Developer Toolset Features](#)

E

Eclipse

- C/C++ source code, [Editing C/C++ Source Code in Eclipse](#)
- Documentation, [Eclipse Documentation](#)
- Help system, [Eclipse Documentation](#)
 - Contents (Help Contents), [Eclipse Documentation](#)
 - Dynamic Help, [Eclipse Documentation](#)
 - Menu (Help Menu), [Eclipse Documentation](#)
 - Workbench User Guide, [Eclipse Documentation](#)
- integrated development environment, [Eclipse User Interface](#)
 - Command Group Availability Tab, [Customize Perspective](#)
 - configuring keyboard shortcuts, [Keyboard Shortcuts](#)
 - Customize Perspective Menu, [Customize Perspective](#)
 - IDE (integrated development environment), [Eclipse User Interface](#)
 - Keyboard Shortcuts Menu, [Keyboard Shortcuts](#)
 - menu (Main Menu), [Eclipse User Interface](#)
 - Menu Visibility Tab, [Customize Perspective](#)
 - perspectives, [Eclipse User Interface](#)
 - Quick Access Menu, [The Quick Access Menu](#)
 - Shortcuts Tab, [Customize Perspective](#)
 - Tool Bar Visibility, [Customize Perspective](#)
 - user interface, [Eclipse User Interface](#)
 - workbench, [Eclipse User Interface](#)
- introduction, [Eclipse Development Environment](#)
- Java Development, [Editing Java Source Code in Eclipse](#)

- libhover plug-in, [libhover Plug-in](#)
- profiling, [Valgrind Plug-in for Eclipse](#)
- projects, [Starting an Eclipse project](#)
 - New Project Wizard, [Starting an Eclipse project](#)
 - technical overview, [Starting an Eclipse project](#)
 - workspace (overview), [Starting an Eclipse project](#)
 - Workspace Launcher, [Starting an Eclipse project](#)
- Quick Access Menu, [The Quick Access Menu](#)
- RPM Building, [Eclipse RPM Building](#)
- User Interface, [Eclipse User Interface](#)
- user interface
 - Console View, [Eclipse User Interface](#)
 - default, [Eclipse User Interface](#)
 - Editor, [Eclipse User Interface](#)
 - Outline Window, [Eclipse User Interface](#)
 - Problems View, [Eclipse User Interface](#)
 - Project Explorer, [Eclipse User Interface](#)
 - quick fix (Problems View), [Eclipse User Interface](#)
 - Tasks Properties, [Eclipse User Interface](#)
 - Tasks View, [Eclipse User Interface](#)
 - tracked comments, [Eclipse User Interface](#)
 - View Menu (button), [Eclipse User Interface](#)
- version, [Red Hat Developer Toolset Features](#)

Editor

- user interface
 - Eclipse, [Eclipse User Interface](#)

elfedit

- features, [New Features](#)

elfutils

- version, [Red Hat Developer Toolset Features](#)

execution (forked)

- GNU debugger, [Forked Execution](#)

F

finish

- tools
 - GNU debugger, [Simple GDB](#)

forked execution

- GNU debugger, [Forked Execution](#)

formatted debugging output

- Python pretty-printers
 - debugging, [Python Pretty-Printers](#)

framework (ftrace)

- profiling
 - ftrace, [ftrace](#)

ftrace

- profiling, [ftrace](#)
- debugfs file system, [ftrace](#)
- documentation, [ftrace Documentation](#)
- framework (ftrace), [ftrace](#)
- usage, [Using ftrace](#)

function tracer

- profiling
- ftrace, [ftrace](#)

fundamental commands

- fundamentals
- GNU debugger, [Simple GDB](#)

fundamental mechanisms

- GNU debugger
- debugging, [GDB](#)

fundamentals

- GNU debugger, [Simple GDB](#)

G**gcc**

- GNU Compiler Collection
- compiling and building, [GNU Compiler Collection \(GCC\)](#)

GDB

- GNU debugger
- debugging, [GDB](#)

Git

- version, [Red Hat Developer Toolset Features](#)

GNOME Power Manager

- libraries and runtime support, [GNOME Power Manager](#)

gnome-power-manager

- GNOME Power Manager
- libraries and runtime support, [GNOME Power Manager](#)

GNU C++ Standard Library

- libraries and runtime support, [The GNU C++ Standard Library](#)

GNU Compiler Collection

- compatibility, [Language Compatibility](#), [Compatibility Changes](#), [Fortran 2003 Compatibility](#), [Fortran 2008 Compatibility](#), [Fortran 77 Compatibility](#), [ABI Compatibility](#), [Debugging Compatibility](#), [Other Compatibility](#)
- compiling and building, [GNU Compiler Collection \(GCC\)](#)
- features, [Status and Features](#), [New Features](#), [Fortran 2003 Features](#), [Fortran 2008 Features](#)
- version, [Red Hat Developer Toolset Features](#)

GNU debugger

- conditional breakpoints, [Conditional Breakpoints](#)
- debugging, [GDB](#)
- documentation, [GDB Documentation](#)
- execution (forked), [Forked Execution](#)
- forked execution, [Forked Execution](#)
- fundamentals, [Simple GDB](#)
 - breakpoint, [Simple GDB](#)
 - commands, [Simple GDB](#)
 - halting an executable, [Simple GDB](#)
 - inspecting the state of an executable, [Simple GDB](#)
 - starting an executable, [Simple GDB](#)
- interfaces (CLI and machine), [Alternative User Interfaces for GDB](#)
- thread and threaded debugging, [Debugging Individual Threads](#)
- tools, [Simple GDB](#)
 - backtrace, [Simple GDB](#)
 - continue, [Simple GDB](#)
 - finish, [Simple GDB](#)
 - help, [Simple GDB](#)
 - list, [Simple GDB](#)
 - next, [Simple GDB](#)
 - print, [Simple GDB](#)
 - quit, [Simple GDB](#)
 - step, [Simple GDB](#)
- usage, [Running GDB](#)
 - debugging a Hello World program, [Running GDB](#)
- variations and environments, [Alternative User Interfaces for GDB](#)

GNU Debugger

- version, [Red Hat Developer Toolset Features](#)

gprof

- features, [New Features](#)

H

halting an executable

- fundamentals
 - GNU debugger, [Simple GDB](#)

helgrind

- tools
 - Valgrind, [Valgrind Tools](#)

help

- tools
 - GNU debugger, [Simple GDB](#)

Help system

- Eclipse, [Eclipse Documentation](#)

host (compile server host)

- compile server
 - SystemTap, [SystemTap Compile Server](#)

Hover Help

- libhover
 - libraries and runtime support, [Setup and Usage](#)

I

IDE (integrated development environment)

- integrated development environment
 - Eclipse, [Eclipse User Interface](#)

indexing

- libhover
 - libraries and runtime support, [libhover Plug-in](#)

inspecting the state of an executable

- fundamentals
 - GNU debugger, [Simple GDB](#)

installation

- debuginfo-packages
 - debugging, [Installing Debuginfo Packages](#)

integrated development environment

- Eclipse, [Eclipse User Interface](#)

interfaces (CLI and machine)

- GNU debugger, [Alternative User Interfaces for GDB](#)

introduction

- compiling and building, [Compiling and Building](#)
- debugging, [Debugging](#)
- Eclipse, [Eclipse Development Environment](#)
- libraries and runtime support, [Libraries and Runtime Support](#)
- profiling, [Profiling](#)
 - SystemTap, [SystemTap](#)

ISO 14482 Standard C++ library

- GNU C++ Standard Library
 - libraries and runtime support, [The GNU C++ Standard Library](#)

ISO C++ TR1 elements, added support for

- GNU C++ Standard Library
 - libraries and runtime support, [GNU C++ Standard Library Updates](#)

J

Java

- libraries and runtime support, [Java](#)

Java Development

- Eclipse, [Editing Java Source Code in Eclipse](#)

K

KDE Development Framework

- libraries and runtime support, [KDE Development Framework](#)

KDE4 architecture

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

kdelibs-devel

- KDE Development Framework
 - libraries and runtime support, [KDE Development Framework](#)

kernel information packages

- profiling
 - SystemTap, [SystemTap](#)

Keyboard Shortcuts Menu

- integrated development environment
 - Eclipse, [Keyboard Shortcuts](#)

KHTML

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

KIO

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

KJS

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

KNewStuff2

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

KXMLGUI

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

L

libraries

- runtime support, [Libraries and Runtime Support](#)

libraries and runtime support

- Boost, [Boost](#)
 - boost-doc, [Boost Documentation](#)
 - documentation, [Boost Documentation](#)
 - message passing interface (MPI), [Boost](#)
 - meta-package, [Boost](#)
 - MPICH2, [Boost](#)
 - new libraries, [Boost Updates](#)

- Open MPI, [Boost](#)
- sub-packages, [Boost](#)
- updates, [Boost Updates](#)

- C++ Standard Library, GNU, [The GNU C++ Standard Library](#)
- compat-glibc, [compat-glibc](#)
- compatibility, [Compatibility](#)
- GNOME Power Manager, [GNOME Power Manager](#)
 - gnome-power-manager, [GNOME Power Manager](#)

- GNU C++ Standard Library, [The GNU C++ Standard Library](#)
 - C++0x, added support for, [GNU C++ Standard Library Updates](#)
 - documentation, [GNU C++ Standard Library Documentation](#)
 - ISO 14482 Standard C++ library, [The GNU C++ Standard Library](#)
 - ISO C++ TR1 elements, added support for, [GNU C++ Standard Library Updates](#)
 - libstdc++-devel, [The GNU C++ Standard Library](#)
 - libstdc++-docs, [GNU C++ Standard Library Documentation](#)
 - Standard Template Library, [The GNU C++ Standard Library](#)
 - updates, [GNU C++ Standard Library Updates](#)

- introduction, [Libraries and Runtime Support](#)
- Java, [Java](#)
 - documentation, [Java Documentation](#)

- KDE Development Framework, [KDE Development Framework](#)
 - Akonadi, [KDE4 Architecture](#)
 - documentation, [kdelibs Documentation](#)
 - KDE4 architecture, [KDE4 Architecture](#)
 - kdelibs-devel, [KDE Development Framework](#)
 - KHTML, [KDE4 Architecture](#)
 - KIO, [KDE4 Architecture](#)
 - KJS, [KDE4 Architecture](#)
 - KNewStuff2, [KDE4 Architecture](#)
 - KXMLGUI, [KDE4 Architecture](#)
 - Phonon, [KDE4 Architecture](#)
 - Plasma, [KDE4 Architecture](#)
 - Solid, [KDE4 Architecture](#)
 - Sonnet, [KDE4 Architecture](#)
 - Strigi, [KDE4 Architecture](#)
 - Telepathy, [KDE4 Architecture](#)

- libhover
 - Code Completion, [Setup and Usage](#)
 - Hover Help, [Setup and Usage](#)
 - indexing, [libhover Plug-in](#)
 - usage, [Setup and Usage](#)

- libstdc++, [The GNU C++ Standard Library](#)
- Perl, [Perl](#)
 - documentation, [Perl Documentation](#)
 - module installation, [Installation](#)
 - updates, [Perl Updates](#)

- Python, [Python](#)
 - documentation, [Python Documentation](#)
 - updates, [Python Updates](#)

- Qt, [Qt](#)
 - documentation, [Qt Library Documentation](#)
 - meta object compiler (MOC), [Qt](#)
 - Qt Creator, [Qt Creator](#)
 - qt-doc, [Qt Library Documentation](#)
 - updates, [Qt Updates](#)
 - widget toolkit, [Qt](#)
- Ruby, [Ruby](#)
 - documentation, [Ruby Documentation](#)
 - ruby-devel, [Ruby](#)

Library and Runtime Details

- NSS Shared Databases, [NSS Shared Databases](#)
 - Backwards Compatibility, [Backwards Compatibility](#)
 - Documentation, [NSS Shared Databases Documentation](#)

libstdc++

- libraries and runtime support, [The GNU C++ Standard Library](#)

libstdc++-devel

- GNU C++ Standard Library
 - libraries and runtime support, [The GNU C++ Standard Library](#)

libstdc++-docs

- GNU C++ Standard Library
 - libraries and runtime support, [GNU C++ Standard Library Documentation](#)

list

- tools
 - GNU debugger, [Simple GDB](#)
 - Performance Counters for Linux (PCL) and perf, [Perf Tool Commands](#)

M

machine interface

- GNU debugger, [Alternative User Interfaces for GDB](#)

mallopt, [mallopt](#)

massif

- tools
 - Valgrind, [Valgrind Tools](#)

mechanisms

- GNU debugger
 - debugging, [GDB](#)

memcheck

- tools
 - Valgrind, [Valgrind Tools](#)

memstomp

- version, [Red Hat Developer Toolset Features](#)

Menu (Help Menu)

- Help system
 - Eclipse, [Eclipse Documentation](#)

menu (Main Menu)

- integrated development environment
 - Eclipse, [Eclipse User Interface](#)

Menu Visibility Tab

- integrated development environment
 - Eclipse, [Customize Perspective](#)

message passing interface (MPI)

- Boost
 - libraries and runtime support, [Boost](#)

meta object compiler (MOC)

- Qt
 - libraries and runtime support, [Qt](#)

meta-package

- Boost
 - libraries and runtime support, [Boost](#)

module installation

- Perl
 - libraries and runtime support, [Installation](#)

module signing (compile server authorization)

- SSL and certificate management
 - SystemTap, [Authorizing Compile Servers for Module Signing \(for Unprivileged Users\)](#)

MPICH2

- Boost
 - libraries and runtime support, [Boost](#)

N**new extensions**

- GNU C++ Standard Library
 - libraries and runtime support, [GNU C++ Standard Library Updates](#)

new libraries

- Boost
 - libraries and runtime support, [Boost Updates](#)

New Project Wizard

- projects
 - Eclipse, [Starting an Eclipse project](#)

next

- tools

- GNU debugger, [Simple GDB](#)

NSS Shared Databases

- Library and Runtime Details, [NSS Shared Databases](#)
- Backwards Compatibility, [Backwards Compatibility](#)
- Documentation, [NSS Shared Databases Documentation](#)

O

objcopy

- features, [New Features](#)

objdump

- features, [New Features](#)

opannotate

- tools
- OProfile, [OProfile Tools](#)

oparchive

- tools
- OProfile, [OProfile Tools](#)

opcontrol

- tools
- OProfile, [OProfile Tools](#)

Open MPI

- Boost
- libraries and runtime support, [Boost](#)

operf

- tools
- OProfile, [OProfile Tools](#)

opgprof

- tools
- OProfile, [OProfile Tools](#)

opreport

- tools
- OProfile, [OProfile Tools](#)

OProfile

- profiling, [OProfile](#)
- documentation, [OProfile Documentation](#)
- usage, [Using OProfile](#)
- tools, [OProfile Tools](#)
 - opannotate, [OProfile Tools](#)
 - oparchive, [OProfile Tools](#)
 - opcontrol, [OProfile Tools](#)
 - operf, [OProfile Tools](#)
 - opgprof, [OProfile Tools](#)

- oprofile, [OProfile Tools](#)

- version, [Red Hat Developer Toolset Features](#)

oprofiled

- OProfile
- profiling, [OProfile](#)

Outline Window

- user interface
- Eclipse, [Eclipse User Interface](#)

P

perf

- profiling
 - Performance Counters for Linux (PCL) and perf, [Performance Counters for Linux \(PCL\) Tools and perf](#)
- usage
 - Performance Counters for Linux (PCL) and perf, [Using Perf](#)

Performance Counters for Linux (PCL) and perf

- profiling, [Performance Counters for Linux \(PCL\) Tools and perf](#)
 - subsystem (PCL), [Performance Counters for Linux \(PCL\) Tools and perf](#)
- tools, [Perf Tool Commands](#)
 - commands, [Perf Tool Commands](#)
 - list, [Perf Tool Commands](#)
 - record, [Perf Tool Commands](#)
 - report, [Perf Tool Commands](#)
 - stat, [Perf Tool Commands](#)
- usage, [Using Perf](#)
 - perf, [Using Perf](#)

Perl

- libraries and runtime support, [Perl](#)

perspectives

- integrated development environment
 - Eclipse, [Eclipse User Interface](#)

Phonon

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

Plasma

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

plug-in for Eclipse

- Autotools
 - compiling and building, [Autotools Plug-in for Eclipse](#)

- profiling
 - Valgrind, [Valgrind Plug-in for Eclipse](#)
- specfile Editor
 - compiling and building, [Eclipse RPM Building](#), [Eclipse Built-in Specfile Editor](#)

pretty-printers

- Python pretty-printers
 - debugging, [Python Pretty-Printers](#)

print

- tools
 - GNU debugger, [Simple GDB](#)

Problems View

- user interface
 - Eclipse, [Eclipse User Interface](#)

Profile As

- Eclipse
 - profiling, [Valgrind Plug-in for Eclipse](#)

Profile Configuration Menu

- Eclipse
 - profiling, [Valgrind Plug-in for Eclipse](#)

profiling

- conflict between perf and oprofile, [Using OProfile](#), [Using Perf](#)
- Eclipse, [Valgrind Plug-in for Eclipse](#)
 - Profile As, [Valgrind Plug-in for Eclipse](#)
 - Profile Configuration Menu, [Valgrind Plug-in for Eclipse](#)
- ftrace, [ftrace](#)
- introduction, [Profiling](#)
- OProfile, [OProfile](#)
 - oprofiled, [OProfile](#)
- Performance Counters for Linux (PCL) and perf, [Performance Counters for Linux \(PCL\) Tools and perf](#)
- SystemTap, [SystemTap](#)
 - DynInst, [DynInst with SystemTap 2.0](#)
- Valgrind, [Valgrind](#)

Project Explorer

- user interface
 - Eclipse, [Eclipse User Interface](#)

projects

- Eclipse, [Starting an Eclipse project](#)

Publican

- Documentation Tools, [Publican](#)
 - Adding media to documentation, [Adding Media to Documentation](#)

- Brands, [Brands](#)
- Building a document, [Building a Document](#)
- Building a website, [Building a Website](#)
- Commands, [Commands](#)
- Create a new document, [Create a New Document](#)
- Files, [Files](#)
- Packaging a publication, [Packaging a Publication](#)
- Publican documentation, [Documentation](#)

Python

- libraries and runtime support, [Python](#)

Python pretty-printers

- debugging, [Python Pretty-Printers](#)

Q

Qt

- libraries and runtime support, [Qt](#)

Qt Creator

- Qt
- libraries and runtime support, [Qt Creator](#)

qt-doc

- Qt
- libraries and runtime support, [Qt Library Documentation](#)

Quick Access Menu

- integrated development environment
- Eclipse, [The Quick Access Menu](#)

quick fix (Problems View)

- user interface
- Eclipse, [Eclipse User Interface](#)

quit

- tools
- GNU debugger, [Simple GDB](#)

R

readelf

- features, [New Features](#)

record

- tools
- Performance Counters for Linux (PCL) and perf, [Perf Tool Commands](#)

report

- tools
- Performance Counters for Linux (PCL) and perf, [Perf Tool Commands](#)

required packages

- compiling and building, [Distributed Compiling](#)
- profiling
 - SystemTap, [SystemTap](#)

requirements

- GNU debugger
 - debugging, [GDB](#)

Ruby

- libraries and runtime support, [Ruby](#)

ruby-devel

- Ruby
 - libraries and runtime support, [Ruby](#)

runtime support

- libraries, [Libraries and Runtime Support](#)

S

scripts (SystemTap scripts)

- profiling
 - SystemTap, [SystemTap](#)

setup

- libhover
 - libraries and runtime support, [Setup and Usage](#)

Shortcuts Tab

- integrated development environment
 - Eclipse, [Customize Perspective](#)

signed modules

- SSL and certificate management
 - SystemTap, [Authorizing Compile Servers for Module Signing \(for Unprivileged Users\)](#)
- unprivileged user support
 - SystemTap, [SystemTap Support for Unprivileged Users](#)

Solid

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

Sonnet

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

specfile Editor

- compiling and building, [Eclipse RPM Building](#), [Eclipse Built-in Specfile Editor](#)

SSL and certificate management

- SystemTap, [SSL and Certificate Management](#)

Standard Template Library

- GNU C++ Standard Library
 - libraries and runtime support, [The GNU C++ Standard Library](#)

starting an executable

- fundamentals
 - GNU debugger, [Simple GDB](#)

stat

- tools
 - Performance Counters for Linux (PCL) and perf, [Perf Tool Commands](#)

step

- tools
 - GNU debugger, [Simple GDB](#)

strace

- version, [Red Hat Developer Toolset Features](#)

Strigi

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

sub-packages

- Boost
 - libraries and runtime support, [Boost](#)

subsystem (PCL)

- profiling
 - Performance Counters for Linux (PCL) and perf, [Performance Counters for Linux \(PCL\) Tools and perf](#)

supported templates

- Autotools
 - compiling and building, [Autotools Plug-in for Eclipse](#)

SVN (see Apache Subversion (SVN))**SystemTap**

- compile server, [SystemTap Compile Server](#)
 - host (compile server host), [SystemTap Compile Server](#)
- profiling, [SystemTap](#)
 - documentation, [SystemTap Documentation](#)
 - DynInst, [DynInst with SystemTap 2.0](#)
 - introduction, [SystemTap](#)
 - kernel information packages, [SystemTap](#)
 - required packages, [SystemTap](#)
 - scripts (SystemTap scripts), [SystemTap](#)
- SSL and certificate management, [SSL and Certificate Management](#)
 - automatic authorization, [Automatic Authorization](#)
 - connection authorization (compile servers), [Authorizing Compile Servers for Connection](#)

- module signing (compile server authorization), [Authorizing Compile Servers for Module Signing \(for Unprivileged Users\)](#)
- unprivileged user support, [SystemTap Support for Unprivileged Users](#)
 - signed modules, [SystemTap Support for Unprivileged Users](#)
- version, [Red Hat Developer Toolset Features](#)

T

Tasks Properties

- user interface
 - Eclipse, [Eclipse User Interface](#)

Tasks View

- user interface
 - Eclipse, [Eclipse User Interface](#)

technical overview

- projects
 - Eclipse, [Starting an Eclipse project](#)

Telepathy

- KDE Development Framework
 - libraries and runtime support, [KDE4 Architecture](#)

templates (supported)

- Autotools
 - compiling and building, [Autotools Plug-in for Eclipse](#)

thread and threaded debugging

- GNU debugger, [Debugging Individual Threads](#)

Tool Bar Visibility

- integrated development environment
 - Eclipse, [Customize Perspective](#)

tools

- GNU debugger, [Simple GDB](#)
- OProfile, [OProfile Tools](#)
- Performance Counters for Linux (PCL) and perf, [Perf Tool Commands](#)
- profiling
 - Valgrind, [Valgrind Tools](#)
- Valgrind, [Valgrind Tools](#)

tracked comments

- user interface
 - Eclipse, [Eclipse User Interface](#)

U

unprivileged user support

- SystemTap, [SystemTap Support for Unprivileged Users](#)

unprivileged users

- unprivileged user support
 - SystemTap, [SystemTap Support for Unprivileged Users](#)

updates

- Boost
 - libraries and runtime support, [Boost Updates](#)
- GNU C++ Standard Library
 - libraries and runtime support, [GNU C++ Standard Library Updates](#)
- Perl
 - libraries and runtime support, [Perl Updates](#)
- Python
 - libraries and runtime support, [Python Updates](#)
- Qt
 - libraries and runtime support, [Qt Updates](#)

usage

- GNU debugger, [Running GDB](#)
 - fundamentals, [Simple GDB](#)
- libhover
 - libraries and runtime support, [Setup and Usage](#)
- Performance Counters for Linux (PCL) and perf, [Using Perf](#)
- profiling
 - ftrace, [Using ftrace](#)
 - OProfile, [Using OProfile](#)
- Valgrind
 - profiling, [Using Valgrind](#)

user interface

- integrated development environment
 - Eclipse, [Eclipse User Interface](#)

V**Valgrind**

- profiling, [Valgrind](#)
 - commands, [Valgrind Tools](#)
 - documentation, [Valgrind Documentation](#)
 - plug-in for Eclipse, [Valgrind Plug-in for Eclipse](#)
 - tools, [Valgrind Tools](#)
 - usage, [Using Valgrind](#)
- tools
 - cachegrind, [Valgrind Tools](#)
 - callgrind, [Valgrind Tools](#)
 - helgrind, [Valgrind Tools](#)
 - massif, [Valgrind Tools](#)
 - memcheck, [Valgrind Tools](#)

- version, [Red Hat Developer Toolset Features](#)

variable tracking at assignments (VTA)

- debugging, [Variable Tracking at Assignments](#)

variations and environments

- GNU debugger, [Alternative User Interfaces for GDB](#)

View Menu (button)

- user interface
 - Eclipse, [Eclipse User Interface](#)

W

widget toolkit

- Qt
 - libraries and runtime support, [Qt](#)

workbench

- integrated development environment
 - Eclipse, [Eclipse User Interface](#)

Workbench User Guide

- Help system
 - Eclipse, [Eclipse Documentation](#)

workspace (overview)

- projects
 - Eclipse, [Starting an Eclipse project](#)

Workspace Launcher

- projects
 - Eclipse, [Starting an Eclipse project](#)